DEVELOPING AN EDUCATIONAL VIDEO GAME FROM ARCHITECTURAL PRECEDENTS

A D.ARCH PROJECT SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF ARCHITECTURE

MAY 2019

By
Alice M. Sandstrom

Thesis Committee:
Hyoung-June Park, Chairperson
Scott Robertson
Jason Steiner

Keywords: Shape Grammar, Video Games, Design Schema

Acknowledgments

I would like to acknowledge my chair and everyone on my committee for all the help and feedback they have provided me throughout this process. You all helped give me a push in the right direction whenever I felt stuck, reorienting me whenever I was lost. I am grateful for your patience and understanding. I would also like to acknowledge my fellow classmates who provided me with ample moral support whenever times were tough.

Abstract

This dissertation proposes the development of an educational video game that teaches its players. This proposed game seeks to address the following two issues/problems within the topic of shape grammar. First, the process of designing with shape grammar requires the designer to first develop shape rules at the start of the design process, opposite to what feels natural, where shape rules are developed through recursive analysis during the design process and applied throughout the remainder of the design process.[1] And second, the higher barrier of entry when it comes to using shape grammar in digital design application makes it difficult to utalize in design education.[2]

Within the proposed game a player is given a variety of 3D spatial puzzles to solve. At the start of each puzzle a player is provided a set of *labeled puzzle pieces* and to solve each of these puzzles a player goes through a process of *instantiation, manipulation,* and *connection* to arrange each of those puzzle pieces within a 3D digital environment. As the player is arranging the provided puzzle pieces, the game provides them meaningful feedback on their choices through *scoring* in addition to tracking their actions as *design rules,* compiling those rules into a *design schema.* The player can go back and explore the *design schema* through a process of *exploration.*

This project looks at the development of a prototype version this proposed game using the Martin House by Frank Lloyd Wright as a basis for a single puzzle level to be implemented. This project concludes with an outline for future development of the proposed game.

---

[1] Thomas Grasl and Athanassios Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 32, no. 2 (2018), https://doi.org/10.1017/S0890060417000506, https://www.cambridge.org/core/article/from-shapes-to-topologies-and-back-an-introduction-to-a-general-parametric-shape-grammar-interpreter/18193A97F837350F38ADD09B2C786763; Hyoung-June Park and

Emmanuel-George Vakaló, "A Form-making Algorithm. Shape Grammar Reversed" (paper presented at the Conference on Computer Aided Architectural Design Futures, 2001); U. Piazzalunga and P. Fitzhorn, "Note on a three-dimensional shape grammar interpreter," *Environment and Planning B: Planning and Design* 25, no. 1 (1998), https://doi.org/10.1068/b250011; M. Tapia, "A visual implementation of a shape grammar system," Article, *Environment & Planning B: Planning & Design* 26, no. 1 (1999), http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=1546802&site=ehost-live; Tomas Trescak, Marc Esteva, and Inmaculada Rodriguez, "A shape grammar interpreter for rectilinear forms," *Computer-Aided Design* 44, no. 7 (2012), https://doi.org/10.1016/j.cad.2012.02.009.

[2] Thomas Fischer and Christiane Herr, "Teaching Generative Design" (paper presented at the The Proceedings of the Fourth International Conference on Generative Art 2001, Milan, Italy, 2001); Mine Özkar, "Visual Schemas: Pragmatics of Design Learning in Foundations Studios," *Nexus Network Journal* 13, no. 1 (2011/04/01 2011), https://doi.org/10.1007/s00004-011-0055-7, https://doi.org/10.1007/s00004-011-0055-7; Bojan Tepavčević and Vesna Stojaković, "Shape grammar in contemporary architectural theory and design," *Facta Universitatis-series: Architecture and Civil Engineering* 10, no. 2 (2012).

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Shape grammar is a means of calculating with shapes by manipulating spatial arrangements and compositions.[1] Shape grammar is also commonly used in architectural analysis of particular styles in order to develop a design language.[2] Shape grammar has been an approach to design education in which conveys tacit knowledge within the field of architectural design. Tacit knowledge in architecture typically is learned through action, which involves designing without explicit awareness of the learning contents themselves.[3] Shape grammar naturally lends itself to this type of education as the shapes used in the shape calculations can be looked at as abstractions of architectural elements.[4] However, the difficulty in the application of shape grammar in the design process makes utilizing shape grammar for this educational purpose difficult, particularly when it comes to beginners.

This difficulty can be explained in part because typically when applying shape grammar within one's design process, the rules are developed first and then the process of *making* the design comes after, which is opposite to what feels natural when designing.[5] The counter-intuitive nature of shape grammar makes it difficult for people to get into initially, and limits its applicability in design education and practice.[6]

This project proposes a simple, 3D puzzle game based on architectural precedents using the *Unity game engine* to address the problems outlined above. Video games are interactive, require no scripting knowledge on the part of the user, are capable of running scripts to analyze what a player is doing while they are doing it, and typically incorporate real-time feedback to the player which judges/score the users' actions (like providing the player a score at the end of a level). [7] The *Unity* engine is by Unity Technologies and supports 2D and 3D games across multiple gaming platforms, including consoles, pcs (laptops and desktop computers), and mobile (smartphones). It was selected in large part due to its flexibility of application, not being too closely tied to any one genre of video game, and its ability and history of being adapted to a large variety of game genres.

Specifically, this project seeks to lay the groundwork for a future video game in which a player would solve a series of spatial puzzles based on a variety of architectural precedents, receiving feedback on their decisions through scoring, as well as being able to both see and explore the various branching decision making paths (*schema*) that they took to arrive where they are within a particular puzzle.

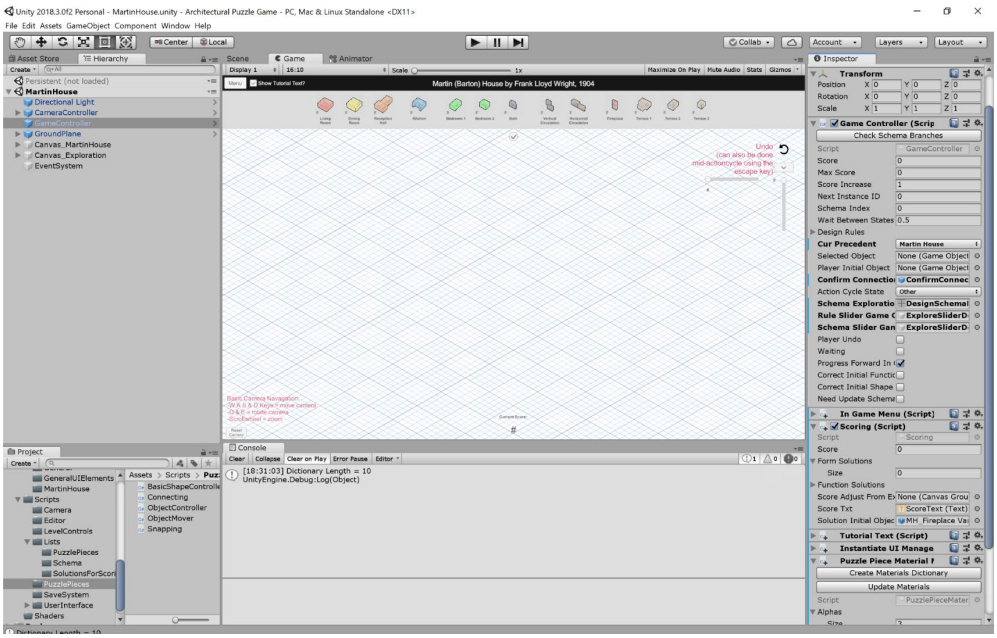


Figure 1-1 Screenshot of Unity Editor

Source: Author

1.1 Document Outline

Chapter 1: Introduction goes over some basic background information about shape grammar, its application as a tool to analyze styles, and its applications in digital design as well as design education. Section 1.3 in this chapter will outline the problem statement this dissertation looks to begin solving and proposes a solution to those problems. This chapter will conclude with some additional information about video games and the *Unity* game engine with the goal of explaining some concepts and terms that the reader might not be familiar.

Chapter 2: Scenarios outlines the process of arriving at the proposed video game solution outlined in chapter one. This process was one that looked at developing a series of different scenarios that looked at ways of merging shape grammar, digital design, and video game technology. Three different scenarios were developed for this project, with the last of the three being divided up into two. This chapter concludes with the thought process behind the selection of one of those scenarios, scenario 3.1 an educational 3D puzzle video game containing puzzles based off architectural precedents, to be used as a basis for the video game prototyped for this project.

Chapter 3: Abstraction Methodology outlines the process of selecting a precedent, the Martin (Barton) House by Frank Lloyd Wright, to use in the initial development for the proposed game and the process by which it and one other precedent, the Villa Savoye by Le Corbusier, where abstracted into massings used for the puzzle pieces. Each abstracted iteration for both precedents were mocked up as either a physical model or in Rhino and tested to see how easy or hard they were to put back together. This chapter concludes with a summary of the different types of shapes that comprise these puzzle pieces and general tips on what was found to work and not work.

Chapter 4: Gameplay outlines the basic gameplay mechanics achieved in this iteration of the proposed game. It describes the actions a player has available to them when solving the puzzle level implemented in the games current build as well as the outcomes of those actions. This chapter will introduce the idea of varying player motivations and this proposed game containing a *library of precedent puzzles* that are hinted at in the current game build, but not fully implemented here, before going into detail about the cycle of player directed *instantiation, manipulation,* and *connection* used to assemble each of the puzzle pieces; how those actions are then scored and tracked by the game; and how a player can go back and change past decisions. This chapter concludes with a section on gameplay difficulty and what aspects of both the *Unity* and *Rhino* builds for these puzzles were seen to impact how easy or hard it was to reassemble and how some of the concerns about difficulty that arose are addressed in the gameplay and set up in the proposed game's iteration developed for this project.

Chapter 5: Implementation goes through and details how the different gameplay mechanics and features were implemented in *Unity.* These features were implemented through a combination of components and objects native to *Unity* or imported to the *Unity* editor from *Rhino* and custom C# scripts written specifically for this project. This chapter will assume that the reader has some basic knowledge of *Unity* and will use a lot of its terminology.

Chapter 6: Discussion & Framework for Future Development outlines some of the takeaways from this prototype build and its development process. It will look at aspects of gameplay that were not able to be implemented within the timeframe and scope of this project but that should be considered when developing the next iteration of the proposed game. The purpose of this chapter is to outline what should come next in terms of the development of an education 3D puzzle game containing

puzzles based off architectural precedents for the purpose of communicating tacit knowledge of architectural design to its players.

1.2 What is Shape Grammar?

Shape grammar is a means of calculating with shapes in order to generate new and unique designs through the application of shape rules. In this regard, shape grammar is a generative design process.[8] The exact definition of what constitutes a shape grammar has changed over time, going through several different iterations, with each subsequent definition becoming more and more ambiguous and broad in its scope.[9] In his paper "*Introduction to Shape and Shape Grammars,"* Stiny (one of the co-creators of shape grammar) defines *shape* to as "a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated Euclidian metric" and *shape grammars* as being comprised of four parts:

1. A finite set of shapes *S*
2. A finite set of symbols *L*
3. A finite set of *shape rules R*. These rules take the form $\alpha \rightarrow \beta$, where α is a labelled shape found in $(S, L)^+$, and β is found in $(S, L)^*$
4. A labeled shape I found in $(S, L)^+$ known as the *initial shape*.[10]

These elements in a shape grammar come together to define a set of shapes and shape rules known as a *language*.[11]

It is important to notice here that the process of designing with shape grammar requires that rules are first generated prior to the generation of the design. This is opposite to what feels natural to do when designing, where the designer first does the *making* and designing and then, through analyzing their work, develops design rules and schema that can be applied in addition future designs or applied in the continuation of the current design in a recursive/iterative process of design and analysis.[12]

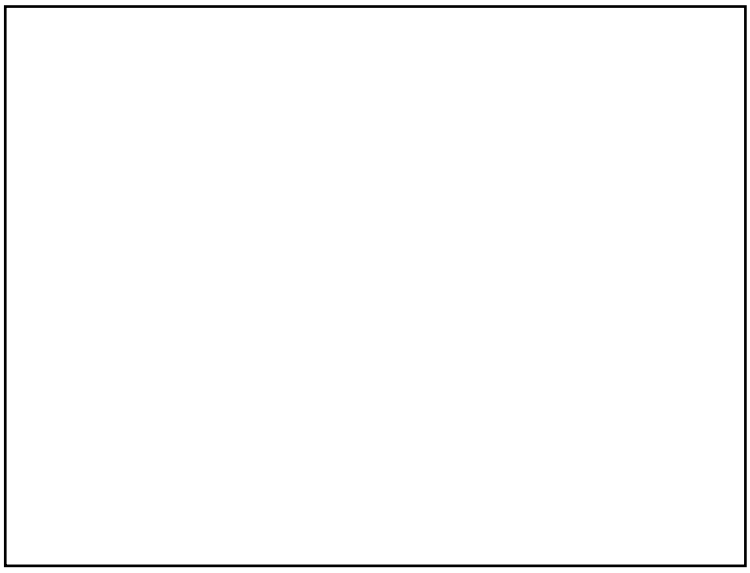


Figure 1-2 Example of a simple shape grammar
Source: "Introduction to Shape and Shape Grammars" by Stiny

In the following subsections, I will broadly go over some of the research that has been done surrounding the use of shape grammar and other similar grammars within the field of design.

1.2.1 Design Computation Process

The idea of viewing design as a solution which is arrived at through a series of computations is not unique to shape grammar and is an approach to solving design problems in terms of its formal components as well as functional, cultural, and political.[13] This project, however, primarily focuses on computations that are done in terms of the formal and functional aspects of design. Spatial grammars deal primarily with computing the formal aspects within a design and provide a designer with a means of bridging the gap between the visual representations of spatial form that are commonly used by designers and the process of computing a solution to a problem using a set of variables and formulae. Of these spatial grammars, shape grammar is the most well-known.[14]

The specifics surrounding the process of computing with shapes and the various factors which one needs to take into consideration when setting up or using shape grammar as a generative design tool is an area of focus for researchers into shape grammar theory. *Computing with Ambiguity* and *Computing with Emergence*, both written by Terry Knight (one of the primary authorities on shape grammars),

deal with the ambiguous nature of shapes and how the unexpected can incorporated into the shape calculation process.[15] In his paper *Weights,* Stiny details out how labels and weights can be applied to shapes within a given set can help augment algebras of shapes.[16]

Ramesh Krishnamurti and Kui Yue's paper on the development of tractable shape grammars details how to go about making shape grammars more conducive to computer applications.[17] Theodora Vardouli examined the process of using shape grammars as it relates to a design's function (as opposed to its simply its pure formal aspect) in her paper *Making Use: Attitudes to Human-Artifact Engagements*.[18]

1.2.2 Other Types of Grammars

As stated before, shape grammar is not the only type of grammar out there within the field of design, and there are other kinds of grammars which have branched off from the core process of shape calculations. These include color grammars, making grammars, and Kindergarten grammars to name a few.[19] Each grammar adds to the areas in which the process of calculation within the field of design can be applied. Color grammar expands the definition of rules to include color, textures, and materiality in addition to the spatial organization examined in traditional shape grammar.[20] Making grammar emphasizes more heavily the process of making objects and physical things as opposed to the more analysis and abstract shape centric position of shape grammar.[21] Kindergarten grammar focuses on applying shape rules to the simple building blocks used in Frobel's Kindergarten method of design education and play.[22]

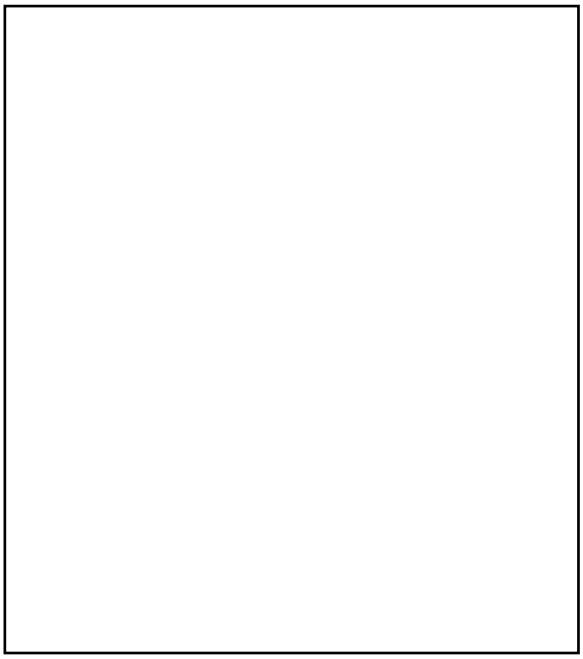


Figure 1-3 Example of Kindergarten Grammar Source: "Kindergarten Grammars: Designing with Frobel's Building Gifts" by Stiny

Another grammar that exists within / adjacent to shape grammar is graph gramma which combines aspects of shape grammar with graph theory. Graph theory is a field of mathematics research which looks to represent and calculate with graphs. In simple terms,

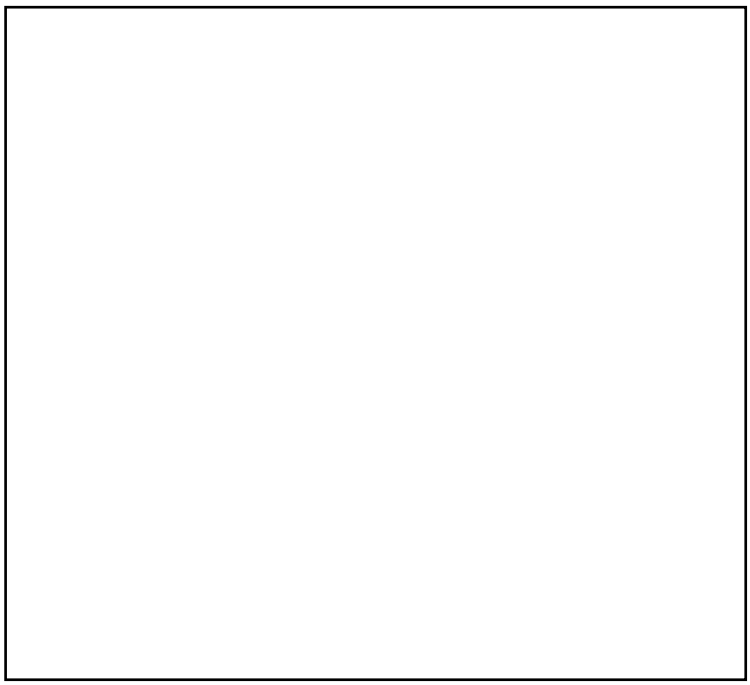


Figure 1-4 Example of Construction a JPG Grammar
Source: "A Justified Plan Graph (JPG) grammar approach to identifying spatial design patterns in an architectural style" by Lee, Ostwald, and Gu

graphs in graph theory are comprised of vertices (dots) and edges (lines which connect the dots).[23] Graph grammars in architectural research are commonly referred to as Justified Plan Graph Grammars (JPGG or JPG Grammar) and are commonly used when looking at digital applications of shape grammar.[24] Another approach to computational design is space syntax, and while not a type of grammar per se, it is none the less worth mentioning here as JPGGs incorporate aspects of Space Syntax). Space syntax is a computational approach that looks at a design's spatial typologies along with its social relations as opposed to shape grammars emphasis on the formal aspects of design.[25] This method of computation is primarily analytical and is typically used to explore the epistemological aspects of design and has more to do with looking at what makes a design meaningful in its functional use beyond a shallow understanding of its formal composition.[26] In their paper, *A Syntactical and Grammatical Approach to Architectural Configuration, Analysis and Generation*, Ju Hyun Lee, Michael Ostwald, and Ning Gu look at the protentional usefulness of combining space syntax with shape grammar using JPG grammars in developing grammars that address both syntactical and formal aspects of design.[27] In their paper, *A Justified Plan Graph (JPG) Grammar Approach to Identifying Spatial Design Patterns in an Architectural Style*, Ju Hyun Lee and Michael Ostwald combine

aspects from shape grammar and space syntax into a justified plan graph grammar.[28]

1.2.3 Design Languages

Sets of shapes and shape rules together form a *language* and can be derived through the analysis of a body of architectural work belonging to a selected architectural style.[29] A decent body of research has done with the aim of defining a particular design language through the analysis of works by specific architects or from specific times in architectural history.

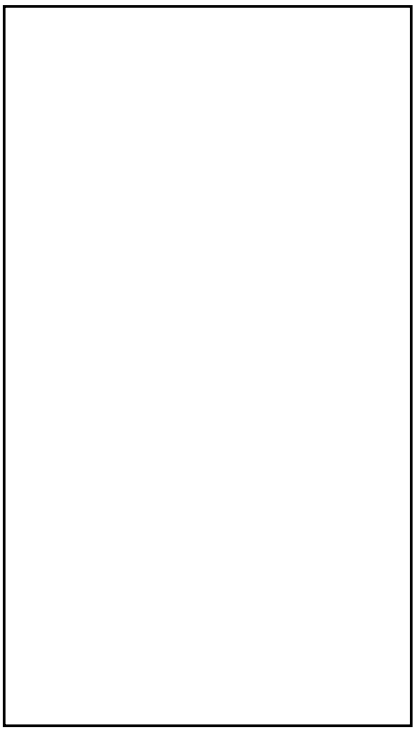


Figure 1-5 Shape Rules for Prairie Style
Source: Koning and Eizenberg

Some of the architectural design languages that have currently been defined using shape grammar thus so far include Cristopher Wren's City Churches, the traditional Bosnian *Hyatt* houses, Siza's houses at Malagueira, Terragni's Cassa Giuliani Frigerio, Frank Lloyd Wright's Prairie Style House, the domestic architecture of Glenn Murcutt, and the Palladian Villa to name a few.[30] Various other design languages outside of architecture have also been defined this way, including wood frame multipurpose chairs, Hepplewhite-style chair backs, Chinese lattice designs, and Islamic patterns and ornamentations.[31]

1.2.4 Application in Architectural Design Software

Shape grammar has not seemed to have found much prevalence within architectural design practice, but there have been recent efforts towards the development of digital shape grammar interpreters. Bojan Tepavčevìć and Vesna Stojakovìĉ, in their paper *Shape Grammar in Contemporary Architectural Theory and Design*, examined how shape grammars have been applied in architectural design, making a note of how the most successful applications were achieved using computer software, but that such applications require the designer to have high levels of programming knowledge.[32] Some of the difficulties around using and embedding shape grammars in Rhino through Grasshopper and the limitations of

utilizing Grasshopper for the purposes of creating and modifying shape grammars to be used in a 3D Rhino model are outlined in the paper *Embedding Garifuna Shape Grammars in Parametric Design Software* by Andino and Sheng-Fen. [33]

Shape grammar has been used to assist in the fabrication of architectural components. Lawrence Sass used shape grammar to divide initial solid shapes into components that could be fabricated by CNC wood routing in order to create house designs that are made from ¾" plywood sheets.[34] Gehry Partners have also used shape grammar algorithms to assist in the fabrication process. In particular, they use shape grammar algorithms to rationalize surface forms for fabrication and construction.[35] Gehry Partners use shape grammar algorithms to figure out how to construct the surfaces of his buildings.[36]

Yue Kui, a software engineer at Microsoft with an architecture background, and Ramesh Krishnamurti have written several papers detailing the development of shape grammars which lend themselves more readily to computer applications, *tractable shape grammars*. Yui Kui's Ph.D. thesis specifically, details how shape grammars can be used as a means of generating the design of interior spaces based upon the exterior constraints of a building.[37] This topic is then further elaborated in several papers both Kui and Krishnamurti have published since 2013.[38]

Generative Geometric Design by Hisserman looks at how shape grammars are applied to solid models, particularly in the area of computer-aided design.[39] A digital shape grammar interpreter known as GRAPE is detailed in the article *From Shapes to Topologies and back; an Introduction to a General Parametric Shape Grammar Interpreter* by Grasl and Economou.[40]

The paper *Using Shape Grammar to Design Ready-Made Housing for Humanized Living - Towards A Parametric-Typological Design Tool* by Linhares and others describes a generative design tool that can be used to generated generate affordable and adaptable housing. [41] Ahmet Emre, Gulen, and Hakan, in their paper *A Digital Tool for Customized Mass Housing Design,* talk about how digital shape grammar tools can be used to generate housing design plans based off client preferences. [42]

Shape grammars application in the field of digital design is not limited to just works of architecture. Bojan Tepavčevìć and Vesna Stojakovìĉ, in their paper *Shape Grammar in Contemporary Architectural Theory and Design* mention how CityEngine generates 3D models automatically through an iterative rule-based refinement process to procedurally generate models of cities using a CGA shape grammar. This

CGA shape grammar language has also been used in urban planning, archeology, digital cultural heritage, and architecture.[43]

1.2.5 Applications in Design Education

Tacit knowledge is knowledge which is hard to transfer through written and spoken language. This form of knowledge is gained through the process of trial and error of trying something and receiving meaningful feedback.[44] This type of knowledge and the process of learning itself is not necessarily a passive process of just receiving knowledge, but an activity. [45] Shape grammar has been one of the methods of conveying/teaching tacit knowledge in architectural design education.

In the 1990s, shape grammar was used to teach architectural design composition to students at MIT, Harvard, UCLA, and Yale, where they would learn a specific architectural design language and then modify the existing to generate unique and personalized variation of an existing language.[46] Several different papers have been written which look into how shape grammar and rule based approaches to design have been used in design education.

Visual Schemas: Pragmatics of Design Learning in Foundations Studios by Özkar looks at how visual schemas can be used in foundation level studio classes to help communicate to new design students how to "help identify, generalize and convey the key aspects of relational and reflective thinking: recursion, seeing emergent shapes as well as parts, boundaries, and relations, and building up variation prior to learning proper shape rules and grammars."[47] Based on this paper it can be deduced that instead of trying to teach design students about shape grammar right from the get go, starting at the schema level might be a better approach.

1.3 Problem Statement & Proposed Solution

Based on my research, the following problems peaked my interest: one, the fact that the process of designing with shape grammar requires the designer to first develop shape rules at the start of the design process, opposite to what feels natural when designing, where shape rules are developed through recursive analysis during the design process and applied throughout the remainder of the design process, where new rules are added as necessary; and two, the higher barrier of entry when it comes to experimenting with shape grammars in digital design makes it harder for

designers to make an initial inquiry into the shape grammar process as well as making it more difficult to utilize in conveying tacit knowledge of architecture within design education.[48]

In order to address the difficulty of utilizing shape grammar in design education of tacit architectural knowledge, it is my goal to lay the groundwork for the development of a video game using the *Unity* game engine where the player would have to solve a series of 3D puzzles based off a variety of architectural precedents.[49] Video games are

Those aspects of video games make them a suitable tool to begin tangibly addressing the problems outlined earlier in this section from a different direction than what most within the field of shape grammar have been doing.

The goal of this project is not to design a completed game, ready to be published and sold, but instead to the development of a first iteration of what a potential video game which addresses these issues might look like as well as to put in some of the core infrastructures such a game would require that are not native to *Unity*. At its core, the goal of this project is to begin implementing for the following, or where that is not possible within the given timeframe, lay the groundwork required:

- Contain a *library* of puzzles based off various architectural precedents
- Player *instantiation* and *manipulation* of puzzle piece game objects
- Player designated *connections* between puzzle piece game objects
- Log the player's actions as *design rules*
- Compile player generated *design rule* to a list of player-generated *design schema*
- Allows the player to go back and explore their branching design choices saved in the *design schema*
- *Score* the player's design based upon how closely it resembles the arrangement and functional connections from the puzzle's precedent

The proposed game is a 3D puzzle game where the player is provided a set of game objects (*labeled puzzle pieces/placeholders*), based on a variety of architectural precedents, to put together through three of the basic transformations (*translation* and *rotation)* of the puzzle pieces to form a design solution. During this

process, the player is provided meaningful feedback on their progress through scoring, done by comparing the results of their actions to the original precedents' arrangement and reasonable function connections derived from the precedent.

Future iterations of the game will contain a *library* of architectural precedents ranging in building typology, style, period, etc. These precedents serve as the basis for each game level's puzzle, determining the set of *puzzle piece game objects* (*shapes*) and *labels* (space functions) the player is provided with at the start of the level, as well as serve as the basis for scoring the player after they solve the level's puzzle. A key source of inspiration for the current form this game has taken is the paper "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses" by Koning and Eizenberg, where they detail a parametric shape grammar which can be used to generate a new architectural design in the prairie style.[50] For this first iteration of the game, only one precedent level is implemented with the understanding that additional precedent levels are necessary in any future iterations. The use of a single precedent for implementation was done to limit the scope of this project to be achievable within the given timeframe.

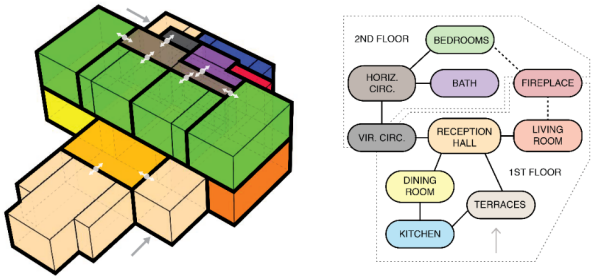


Figure 1-6 Original Arrangement (left) & Functional Diagram (right) for Martin House
Source: Author

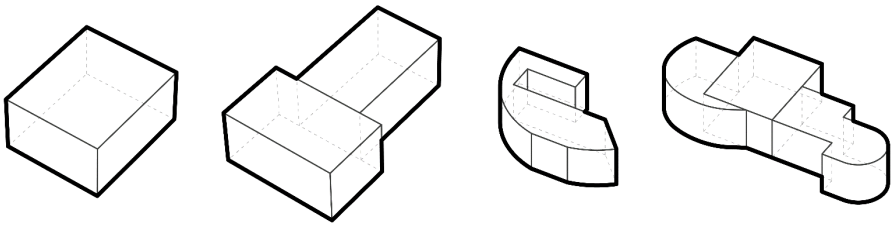


Figure 1-7 Shape Type Examples (From left to right: basic, compound, unique, & unique compound)
Source: Author

Puzzle piece game objects are *shapes* derived from the abstraction of a precedent's layout and act as placeholders for architectural elements. There are four main categories of *shapes*: *basic*, which are simple cuboids; *compound*, which are comprised of a series of basic geometric forms which act as the components of the placeholder; *unique*, which consist of non-cuboid forms; and *unique compound*,

which are identical to *compound shapes* but are comprised of both basic and unique geometric forms. These four types were the result of a series of iterative abstractions for the precedents selected for the games initial prototyping. The image below shows the placeholders from an abstracted layout of one of the selected precedents. The abstraction process itself is gone over in greater depth in Chapter 3: Abstraction Methodology.

Figure 1-8 Abstracted Layout of Martin House and its 3D Placeholders
Source: Author

Alongside the abstraction of the formal aspects of a precedent design, the connections between different functions assigned to each abstracted *puzzle piece game object* are articulated using bubble diagrams, similar to JPGs. These functional sequences are one of the scoring criteria in addition to the comparison of arrangements. The

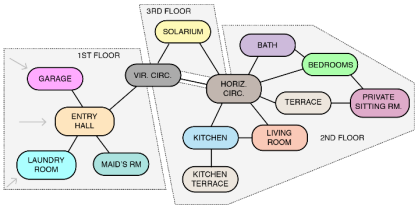


Figure 1-9 Connections Between Functions for Villa Savoye
Source: Author

scoring criteria serves as a means of providing the player with meaningful syntax during gameplay by maintaining a level of design freedom in the step-by-step process of assembling of provided *puzzle pieces* while providing meaningful feedback for each *puzzle piece*.

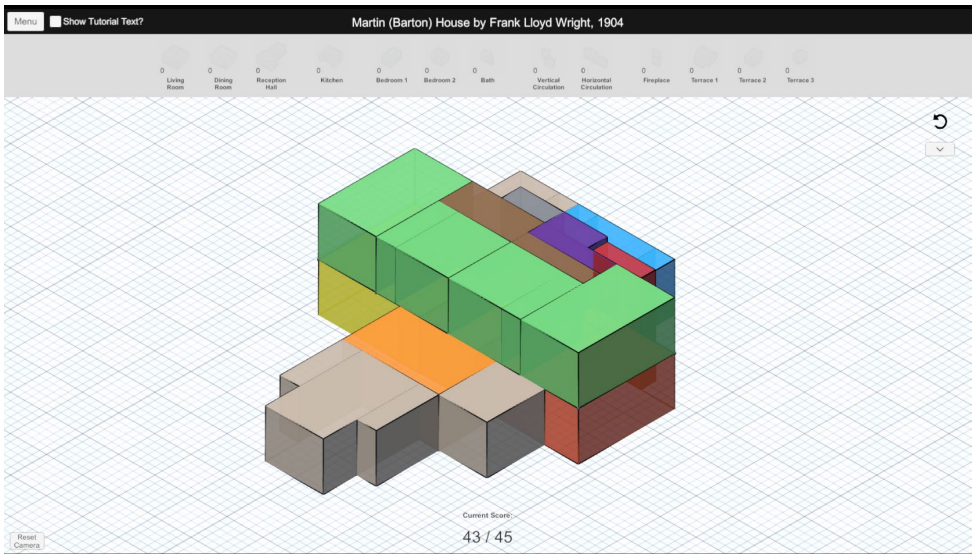


Figure 1-10 Screenshot from Current Iteration of Proposed Game

Source: Author

1.4 Unity

As mentioned in the previous section, current development has taken place using the *Unity* game engine. Because my project touches on aspects of game development which the reader may or may not be familiar with, I will briefly explain some elements of game design and terms used when developing with *Unity*.

1.4.1 What is a Game Engine & Why Unity?

A game engine provides a framework for game development and many different types of game engines exist with varying degrees of specialization. Some games like *The Witcher 3* by CD Project Red use in-house game engines developed by the same game studio, while other games, like *Pillars of Eternity* by Obsidian use a third-party engine. While there are advantages to using a highly tailored engine, creating a game engine from scratch can take thousands of hours. Third-party game engines like *Unity* are useful in that they significantly reduce development time by

already having coding in place to handle some of the basic aspects of game development, like the graphics, audio, and logic.[51]

The *Unity* game engine was selected for several reasons. It is free to use, provided you are not generating revenue, and has a large volume of online resources available to help people learn how to develop with the engine. It is also able to support user-generated scripts using languages like C# and Java. All of this has made *Unity* known for being beginner friendly while still being able to create sophisticated games in a large variety of video game genres.[52] It is for these reasons that *Unity* was selected to be used for this project.

1.4.2 Unity Terminology

In this subsection, I will cover terms and concepts important when using the *Unity Editor*. These ideas are used primarily in Chapter 5: Implementation & Chapter 6: Discussion & Framework for Future Development.

Game Object: Game Objects are the fundamental object in *Unity*. Everything in the game, all the parts of the game, are essentially Game Objects and can include things like characters, collectibles, lights, cameras, special effects, and UI elements. On their own, Game Objects do not do anything; instead they have various components assigned to them that give each Game Object its properties and tell it what functions to perform.[53]

MonoBehaviour: the base class used in *Unity.* It is the class that all C# scripts made in *Unity* inherit from by default and contains key methods for game play like *Start ( )* and *Update ( )* which are key for creating gameplay.[54]

Parenting: a concept in unity where several game objects can be grouped together with one *parent object* and several *child objects* or *children.* These *child game objects* are also able to have their own nested *parent-child objects.* Parenting is useful when setting up game objects as the *child objects* will inherit the movement and rotation form their *parent.*[55] This idea of nesting game objects within other game objects can also be done using prefabs, where one prefab can be nested within another prefab.[56]

Prefabs: a prefab is a reusable asset that saves a Game Object's components, property values, and child Game Objects. Prefabs can be instantiated within any loaded scene in the game through the asset file assigned to the particular project.[57]

Quaternion: used to represent rotations and have components (x, y, z, w).[58]

Raycasts: a method for detecting where to send input events from the Event System based on a given screen space position. There are different types of raycasts, however, this project only uses two of them: one for detecting UI elements and on for 3D physics elements.[59]

Scene: a scene contains the environments and menus of a game. Scenes are where the environments, obstacles, and decorations are placed, and functions as the "place" a player goes through. For example, each level within the game that contains a puzzle for the player to solve refers to a specific scene which contains the parameters and game objects relating to that puzzle.[60]

Tag: "A Tag is a reference word which you can assign to one or more GameObjects […] Tags help you identify GameObjects for scripting purposes. They ensure you don't need to manually add GameObjects to a script's exposed properties […] Tags are useful for triggers in Collider control scripts; they need to work out whether the player is interacting with an enemy, a prop, or a collectable, for example."[61]

Transforms: a transform is a component assigned to each game object. It tells the game object what its position, rotation, scale, and parenting state is currently set to. Every game object comes with this component already assigned, and it can never be removed.[62]

Trigger Collider: Colliders are used for physics collisions and are typically approximations of the mesh shape for a game object and are invisible to the player. Colliders can come in different types, like box collider and sphere collider. Trigger collider will refer to any collider that is told to only detect when another collider enters its space without creating a collision through the *Is Trigger* property of the Collider component. These colliders are used to call certain methods in the object's scripts which are inherited from the *MonoBehaviour* class.[63]

Update & FixedUpdate: Update is a method which is called every frame provide the scripting component is enabled. FixedUpdate is similar to update except it is frame-rate independent and is typically used to handle physics calculations.[64]

[1] G. Stiny, "Introduction to shape and shape grammars," *Environment and Planning B: Planning and Design* 7, no. 3 (1980), https://doi.org/10.1068/b070343; George Stiny, *Shape : talking about seeing and doing* (Cambridge, MA: Massachusetts Institute of Technology, 2006).

[2] H. Buelinckx, "Wren's Language of City Church Designs: A Formal Generative Classification," *Environment and Planning B: Planning and Design* 20, no. 6 (1993), https://doi.org/10.1068/b200645; Birgul Colakoglu, "An Informal Shape Grammars for Interpolations of Traditional Bosnian Hayat Houses in a Contemporary Context" (paper presented at the Generative Art 2002: 5th International Generative Art Conference, Milan, Italy, 2002); José Pinto Duarte, "Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira," *Environment and Planning B: Planning and Design* 32, no. 3 (2005/06/01 2005), https://doi.org/10.1068/b31124, https://doi.org/10.1068/b31124; U. Flemming, "The Secret of the Casa Giuliani Frigerio," *Environment and Planning B: Planning and Design* 8, no. 1 (1981), https://doi.org/10.1068/b080087; Hanson, N. L. R. and Radford, and Antony D, *On Modelling the Work of the Architect Glenn Murcutt* (1986); H. Koning and J. Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses," *Environment and Planning B: Planning and Design* 8, no. 3 (1981/09/01 1981), https://doi.org/10.1068/b080295, http://journals.sagepub.com/doi/abs/10.1068/b080295; Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Ju Hyun Lee, Michael J. Ostwald, and Ning Gu, "A Justified Plan Graph (JPG) grammar approach to identifying spatial design patterns in an architectural style," *Environment and Planning B: Urban Analytics and City Science* 45, no. 1 (2018), https://doi.org/10.1177/0265813516665618; G. Stiny and W. J. Mitchell, "The Palladian Grammar," *Environment and Planning B: Planning and Design* 5, no. 1 (1978), https://doi.org/10.1068/b050005; Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[3] Donald A. Schön, "The reflective practitioner : how professionals think in action," (New York: New York : Basic Books, 1983).

[4] Alice Sandstrom and Hyoung-June Park, "Reflection in Action: An Educational Indie Video Game with Design Schema" (CAADRIA, Wellington, New Zealand, 2019).

[5] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Park and Vakaló, "A Form-making Algorithm. Shape Grammar Reversed."; Piazzalunga and Fitzhorn, "Note on a three-dimensional shape grammar interpreter."; Tapia, "A visual implementation of a shape grammar system."; Trescak, Esteva, and Rodriguez, "A shape grammar interpreter for rectilinear forms."

[6] Fischer and Herr, "Teaching Generative Design."

[7] Sebastian Deterding et al., "From game design elements to gamefulness: defining "gamification"" (paper presented at the 15th International Academic MindTrek Conference, 2011).

[8] Stiny, "Introduction to shape and shape grammars."; Stiny, *Shape : talking about seeing and doing*.

[9] Stiny, "Introduction to shape and shape grammars."; Yue Kui, "Computation-friendly shape grammars with application to determining the interior layout of buildings from image data" (Ph.D. Disseration, Carnegie Mellon University, 2009), http://eres.library.manoa.hawaii.edu/login?url=https://search-proquest-com.eres.library.manoa.hawaii.edu/docview/304862081?accountid=27140 (33824443); Stiny, "Introduction to shape and shape grammars."

[10] Stiny, "Introduction to shape and shape grammars."

[11] Stiny, "Introduction to shape and shape grammars."

[12] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Park and Vakaló, "A Form-making Algorithm. Shape Grammar Reversed."; Piazzalunga and Fitzhorn, "Note on a three-dimensional shape grammar interpreter."; Tapia, "A visual implementation of a shape grammar system."; Trescak, Esteva, and Rodriguez, "A shape grammar interpreter for rectilinear forms."

[13] William J. Mitchell, *The logic of architecture : design, computation, and cognition* (Cambridge, Mass.: Cambridge, Mass. : MIT Press, 1990); Christopher Alexander, *Notes on the Synthesis of Form* (Harvard University Press, 1964).

[14] Kui, "Computation-friendly shape grammars with application to determining the interior layout of buildings from image data."

[15] Terry Knight, "Computing with ambiguity," Article, *Environment & Planning B: Planning & Design* 30, no. 2 (2003), http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=9592105&site=ehost-live; Terry Knight, "Computing with emergence," Article, *Environment & Planning B: Planning & Design* 30, no. 1 (2003), http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=9683517&site=ehost-live.

[16] G. Stiny, "Weights," *Environment and Planning B: Planning and Design* 19, no. 4 (1992), https://doi.org/10.1068/b190413.

[17] Ramesh Krishnamurti and Kui Yue, "Developing a tractable shape grammar," Article, *Environment & Planning B: Planning & Design* 42, no. 6 (2015), https://doi.org/10.1177/0265813515610673, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110826658&site=ehost-live.

[18] Theodora Vardouli, "Making use: Attitudes to human-artifact engagements," Article, *Design Studies* 41 (2015), https://doi.org/10.1016/j.destud.2015.08.002, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110741766&site=ehost-live.

[19] Terry Knight and George Stiny, "Making grammars: From computing with shapes to computing with things," Article, *Design Studies* 41 (2015), https://doi.org/10.1016/j.destud.2015.08.006, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110741765&site=ehost-live; T. W. Knight, "Color grammars: designing with lines and colors," *Environment and Planning B: Planning and Design* 16, no. 4 (1989), https://doi.org/10.1068/b160417; G. Stiny, "Kindergarten grammars: designing with Froebel's building gifts," *Environment and Planning B: Planning and Design* 7, no. 4 (1980), https://doi.org/10.1068/b070409.

[20] Knight, "Color grammars: designing with lines and colors."

[21] Knight and Stiny, "Making grammars: From computing with shapes to computing with things."

[22] Stiny, "Kindergarten grammars: designing with Froebel's building gifts."

[23] "Graph Theory," Descrete Mathmatics: an Open Introduction (Website), 2016, accessed January 21, 2019, http://discrete.openmathbooks.org/dmoi2/ch_graphtheory.html.

[24] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Lee, Ostwald, and Gu, "A Justified Plan Graph (JPG) grammar approach to identifying spatial design patterns in an architectural style."; Ji-Hyun Lee, Michael J. Ostwald, and Ning Gu, "A Combined Plan Graph and Massing Grammar Approach to Frank Lloyd Wright's Prairie Architecture," *Nexus Network Journal* 19, no. 2 (22 February 2017 2017).

[25] Bill Hillier and Julienne Hanson, *The social logic of space* (Cambridge: Cambridge : Cambridge University Press, 1984).

[26] Ju Hyun Lee, Michael J. Ostwald, and Ning Gu, "A syntactical and grammatical approach to architectural configuration, analysis and generation," Article, *Architectural Science Review* 58, no. 3 (2015), https://doi.org/10.1080/00038628.2015.1015948, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=102747902&site=ehost-live.

[27] Lee, Ostwald, and Gu, "A syntactical and grammatical approach to architectural configuration, analysis and generation."

[28] Lee, Ostwald, and Gu, "A Justified Plan Graph (JPG) grammar approach to identifying spatial design patterns in an architectural style."

[29] Stiny, "Introduction to shape and shape grammars."

[30] Buelinckx, "Wren's Language of City Church Designs: A Formal Generative Classification."; Colakoglu, "An Informal Shape Grammars for Interpolations of Traditional Bosnian Hayat Houses in a Contemporary Context."; Duarte, "Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira."; Flemming, "The Secret of the Casa Giuliani Frigerio."; Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Lee, Ostwald, and Gu, "A Justified Plan Graph (JPG) grammar approach to identifying spatial design patterns in an architectural style."; Stiny and Mitchell, "The Palladian Grammar."; Stiny, "Weights."

[31] Ji-Hyun Lee et al., "A formal approach for the interpretation of cultural content(s): evolution of a Korean traditional pattern, Bosangwhamun" (paper presented at the CAAD's New Frontiers: Proceedings of the 15th International Conference on Computer-Aided Architectural Design Research in Asia, Hong Kong SAR, 2010 2010); T. Weissman Knight, "The Generation of Hepplewhite-Style Chair-Back Designs," *Environment and Planning B: Planning and Design* 7, no. 2 (1980), https://doi.org/10.1068/b070227; G. Stiny, "Ice-Ray: A Note on the Generation of Chinese Lattice Designs," *Environment and Planning B: Planning and Design* 4, no. 1 (1977), https://doi.org/10.1068/b040089; Sehnaz Cenani and G. Cagdas, *Shape Grammar of Geometric Islamic Ornaments* (2013); S. Cenani and G. Cagdas, "A Shape Grammar Study: Form Generation with Geometric Islamic Patterns" (paper presented at the Generative Art Conference, Milan, Italy, 2007); Sara Garcia and Luis Romao, *Style and Type in a Generic Shape Grammar: The Case of Multipurpose Chairs* (2016).

[32] Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[33] Dulce Andino and Chien Sheng-Fen, "Embedding Shape Grammars in a Parametric Design Software" (paper presented at the Knowledge-based Design - Proceedings of the 17th Conference of the Iberoamerican Society of Digital Graphics, Valparaiso, Chile, 2013 2013).

[34] Lawrence Sass, *Synthesis of design production with integrated digital fabrication*, vol. 16 (2007); Lawrence Sass, "Wood Frame Grammar: CAD Scripting a Wood Frame House" (CAAD Futures Conference, Vienna, 2005).

[35] Dennis R. Shelden, "Digital Surface Representation and the Constructibility of Gehry's Architecture" (Ph.D in Architecture: Design and Computation Thesis (Ph.D), Massachusetts Institute of Technology, 2002); Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[36] Shelden, "Digital Surface Representation and the Constructibility of Gehry's Architecture."

[37] Kui, "Computation-friendly shape grammars with application to determining the interior layout of buildings from image data."

[38] R. Krishnamurti and C. F. Earl, "Shape recognition in three dimensions," *Environment and Planning B: Planning and Design* 19, no. 5 (1992), https://doi.org/10.1068/b190585; Krishnamurti and Yue, "Developing a tractable shape grammar."; Kui, "Computation-friendly shape grammars with application to determining the interior layout of buildings from image data."; Yue Kui and Ramesh Krishnamurti, "Tractable shape grammars," Article, *Environment & Planning B: Planning & Design* 40, no. 4 (2013), https://doi.org/10.1068/b38227, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=90116960&site=ehost-live; Kui and Krishnamurti, "Tractable shape grammars."; Yue Kui and Ramesh Krishnamurti, "A paradigm for interpreting tractable shape grammars," Article, *Environment & Planning B: Planning & Design* 41, no. 1 (2014), https://doi.org/10.1068/b39107, http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=94594680&site=ehost-live.

[39] L. Heisserman, "Generative geometric design," *Computer Graphics and Applications, IEEE* 14, no. 2 (1994), https://doi.org/10.1109/38.267469, http://ieeexplore.ieee.org/document/267469/?reload=true.

[40] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."

[41] Bruna Linhares et al., "Using Shape Grammar to design ready-made housing for humanized living. Towards a parametric-typological design tool" (paper presented at the Augmented Culture: Proceedings of the 15th Iberoamerican Congress of Digital Graphics, Santa Fe, Argentina, 2011 2011).

[42] Dincer Ahmet Emre, Çağdaş Gülen, and Tong Hakan, "A Digital Tool for Customized Mass Housing Design" (paper presented at the Fusion, Proceedings of the 32nd International Conference on Education and research in Computer Aided Architectural Design in Europe, Newcastle upon Tyne, UK, 2014 2014).

[43] Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[44] Schön, "The reflective practitioner : how professionals think in action."

[45] Fischer and Herr, "Teaching Generative Design."

[46] Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[47] Özkar, "Visual Schemas: Pragmatics of Design Learning in Foundations Studios."

[48] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Park and Vakaló, "A Form-making Algorithm. Shape Grammar Reversed."; Piazzalunga and Fitzhorn, "Note on a three-dimensional shape grammar interpreter."; Sandstrom and Park, "Short Reflection in Action: An Educational Indie Video Game with Design Schema."; Tapia, "A visual implementation of a shape grammar system."; Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."; Trescak, Esteva, and Rodriguez, "A shape grammar interpreter for rectilinear forms."

[49] Deterding et al., "From game design elements to gamefulness: defining "gamification"."; Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[50] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[51] "Unity Technologies logo.svg," 2011, accessed 11/25/2018, https://commons.wikimedia.org/wiki/File:Unity_Technologies_logo.svg; TheHappieCat, "How Game Engines Work!," (September 7 2015), Video file. https://www.youtube.com/watch?v=DKrdLKetBZE.

[52] TheHappieCat, "How Game Engines Work!."

[53] "GameObjects," Manual (Webpage), Unity Technologies, updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/GameObjects.html.

[54] "Monobehaviour," Scripting API, Unity Technologies, updated March, 2018, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

[55] "The Hierarchy Window," Manual, Unity Technologies, updated March 2018, accessed July 31, 2019, https://docs.unity3d.com/Manual/Hierarchy.html.

[56] "Adding a Nested Prefab in Prefab Mode," Manual, Unity Technologies, updated March 2018, accessed March 20, 2019, https://docs.unity3d.com/Manual/NestedPrefabs.html.

[57] "Prefabs," Manual (Webpage), Unity Technologies, updated July 31, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Prefabs.html.

[58] "Quaternion," Scripting API, Unity Technologies, updated March, 2018, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/Quaternion.html.

[59] "Raycasts," Manual (Webpage), Unity Technologies, updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Raycasters.html.

[60] "Scenes," Manual (Webpage), Unity Technologies, updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/CreatingScenes.html.

[61] "Tag," Manual (Webpage), Unity Technologies, updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Tags.html.

[62] "Transform," Manual (Webpage), Unity Technologies, updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/class-Transform.html

[63] "Monobehaviour."; "Colliders," Manual (Webpage), Unity Technologies, updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/CollidersOverview.html.

[64] "Monobehaviour."

Chapter 2: Scenarios

How did this dissertation project go from applications of shape grammar to video game? It this chapter I will go over how it is I arrived at the idea of developing the video game which serves as my dissertation project. Early on in the project, I developed a series of scenarios looking at potential solutions/ideas of how to use shape grammar in digital technology outside of the existing CAD programs. Scenarios are a useful tool when it comes to the development of software and are used to describe the services that the software would provide in addition to the overall context that said software finds itself. It is a way of looking at, thinking about, and describing issues revolving around how a user would use a piece of software and what kinds of user actions the software would have to support. This understanding is critical because, computer systems are not ever ends in and of themselves and are always made be used for something. This use is embedded in and meaningful because of specific use case situations.[1]

The first section of this chapter will summarize the work I had done relating to this project prior to the start of the Fall 2018 semester. It can be thought of as an exploration of the potential feasibility into using the *Unity* engine to develop an application which can explore the use shape grammar within the design process. I will then detail how I went about deciding on the specific form my dissertation project has taken: an educational puzzle game based on existing architectural precedents based on an initial starting premise of wanting to explore the potential of utilizing the interactive nature of video games in using shape grammar within digital assisted design.

2.1 Preliminary Video Game Application Exploration

My initial research looked to address the learning curve and difficulty I saw when trying to apply shape grammar using computer-aided-design applications due to the level of scripting knowledge that one needed to have beforehand.[2] To address this, I began looking into the feasibility of developing a computer application which would allow for the user to experiment with shape grammar in design without requiring them to have any scripting knowledge. This application would take the form of a video game as they do not require any scripting knowledge on the player's part and are an interactive media. This initial exploration was done using the *Unity* game engine for the reasons detailed in section 1.4.

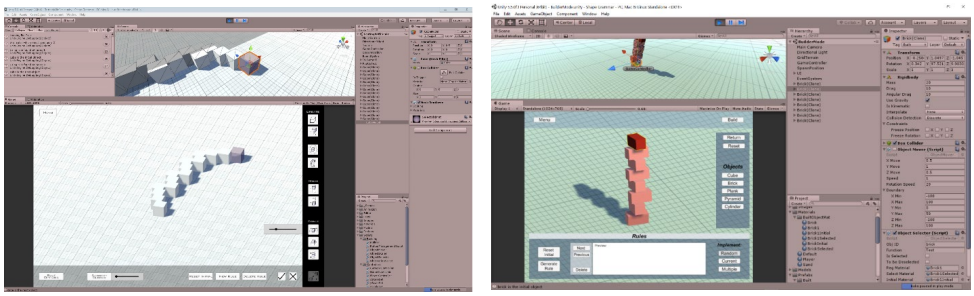


Figure 2-1 Screenshots of some of the initial video game development

Source: Author

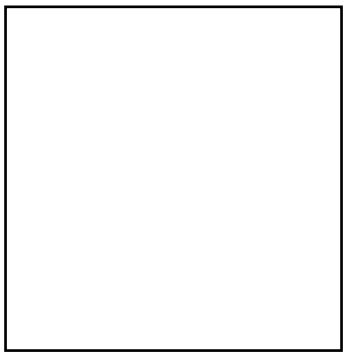


Figure 2-2 Kindergarten Grammar Example

Source: Stiny

The proposed game would allow the player to experiment with shape grammar which are similar in nature to the Kindergarten grammars described by Stiny in his article "Kindergarten Grammars: Designing with Froebel's Building Gifts".[3] Kindergarten grammar was used as the backbone for this initial exploration due to its potential for application within the massing stage of design as well as its relatively simple 3D geometric form consisting of only a few morphological transformations. This limited the amount of geometric complexity that would need to be accounted for within the game's code, thereby reducing the amount of code required making it easier for a beginner in game development like myself to work on. Th e following storyboards detail some of the progress made in this very early inquiry:

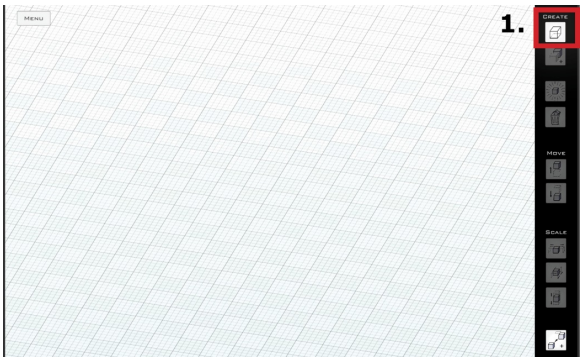


Figure 2-3 Preliminary Storyboard Panel 1

Source: Author

1. Game objects are instantiated within the scene (the game world) using a button on the side panel (in this case a basic cube shape is selected to be instantiated)

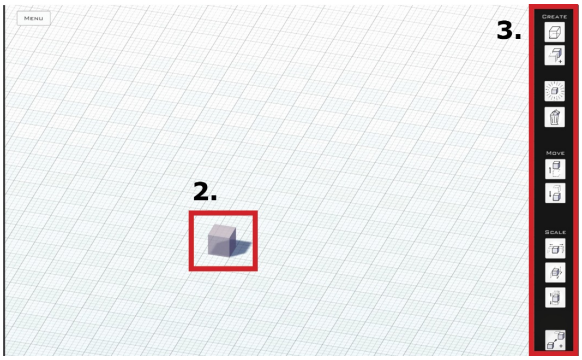


Figure 2-4 Preliminary Storyboard Panel 2

Source: Author

2. When a new game object is instantiated, it is automatically selected, allowing the player to move it around the scene using the WASD keys and changing its color to indicated to the player that it is selected.
3. When an object is selected, buttons that correspond to commands that require an object the scene to be selected are activated and interactable.

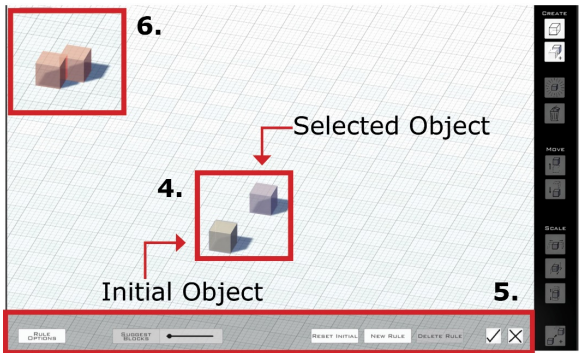


Figure 2-5 Preliminary Storyboard Panel 3

Source: Author

4. When the player instantiates a new object while still having an object within the scene selected, the selected object is deselected and becomes the initial object, changing its color to convey this to the player. The newly instantiated object is then selected.
5. If two or more objects exist within the current scene, the rule panel at the bottom is activated.
6. When two or more objects overlap, they are highlighted and the player is unable to deselect the selected object until the objects are no longer overlapping.

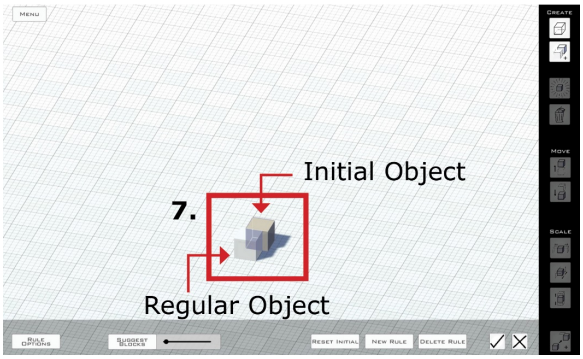


Figure 2-6 Preliminary Storyboard Panel 4

Source: Author

7. Once the player has moved the selected object to where they want it, they can deselect it by using the escape key. If there is an initial object within the scene, a script within the game files runes a method which logs the offset position and rotation between the two shapes to a list of rules.

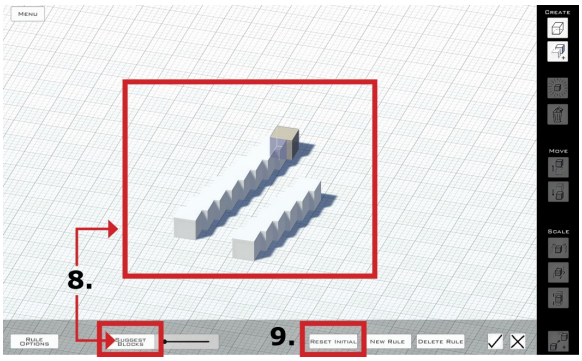


Figure 2-7 Preliminary Storyboard Panel 5

Source: Author

8. Once a rule is saved, the player can then place a new game object in the scene using the transforms saved within the rule.
9. The player can also choose to clear the initial object, meaning that no new rules will be saved until a new initial object has been designated. This is done using the "reset initial" button located in the rule panel at the bottom of the screen.

After working on this idea for a while I felt like I had hit a brick wall and that my project had moved too far away from architectural design for my comfort and as such I felt like I needed rethink what I wanted this project to actually be in the end. Due to its lack luster results and overall lack of purpose, this project would dropped. It is worth noting though as some of the aspects from endeavor did make it over into the current iteration of this dissertation project.

2.2 Scenarios: Overview

After taking some time to think about where I wanted to go with my dissertation topic it was decided at the beginning of the Fall 2018 semester that I

would take a step back from what I had been working on up till this point and reevaluate the purpose of my project – what will the user get out of it.

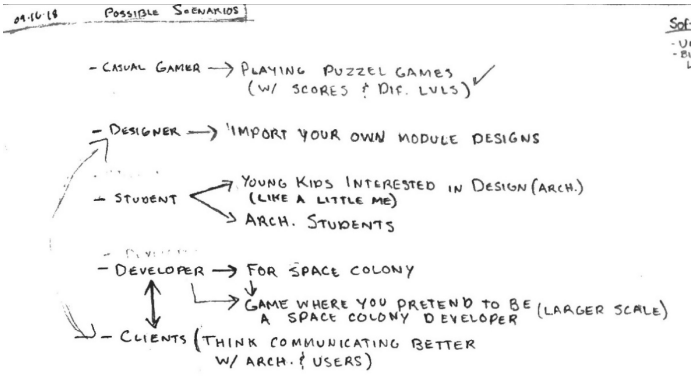


Figure 2-8 Scenario Brainstormed Ideas

Source: Author

Using this brainstormed list, I began fleshing out three different scenarios. I started by defining a user, their purpose for using the proposed application, and what their overall motivation while using the application would be. Once I had those defined, I created a storyboard which would explore step-by-step actions the player would perform, how they would use the proposed application. From the storyboard, a simplified user-action-cycle flow chart was created to outline the basic types of actions a user would make in the order they would make them. The three scenarios are as follow:

- Scenario 1: "Tetris-Like" Puzzle Game
- Scenario 2: Modular System Visualization Tool
- Scenario 3: Precedent Game

Scenario 1 and 2 were created first, after which scenario 3 was created as a variation of scenario 1. Scenario 3 ended up being divided into two different sub-scenarios:

- Scenario 3.1: Addition
- Scenario 3.2: Division

Of the four different scenarios that were created, Scenario 3: Precedent Game version 3.1: Addition was selected as the proposed solution to the problem statement outlined in chapter 1. Figure 2-9 outlines this process of scenario refinement leading to the selection of scenario 3.1 – each of the scenarios created for this project are detailed in sections 2.3 through 2.5.

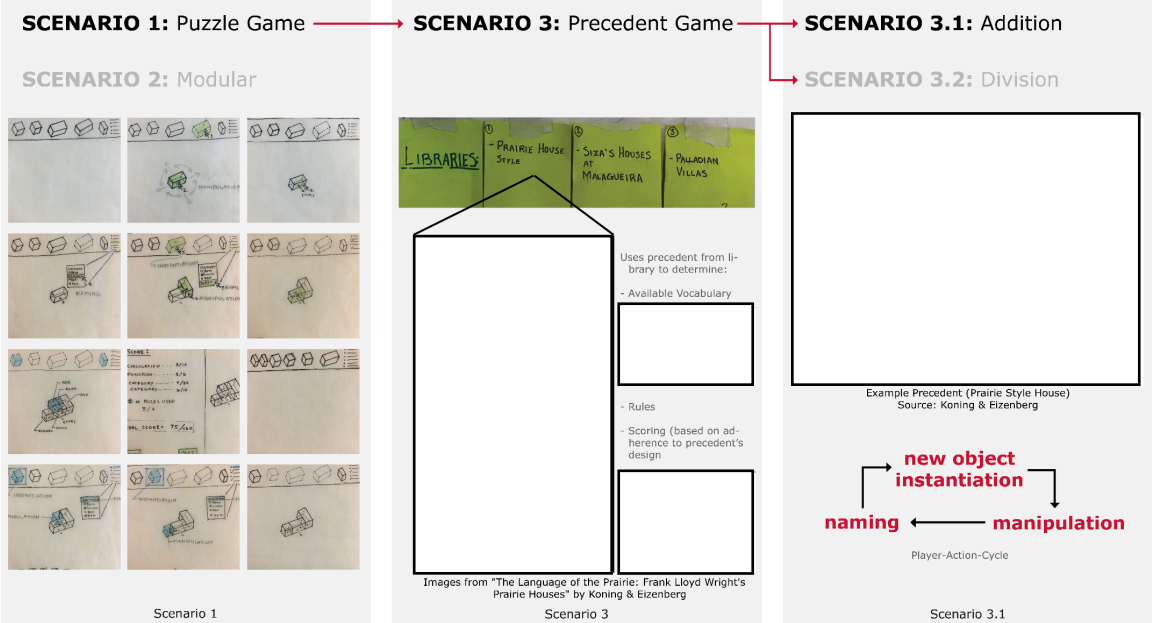


Figure 2-9 Scenario Selection Overview

Source: Author with parts by Koning and Eizenberg[4]

2.3 Scenario 1: "Tetris-Like" Puzzle Game

This scenario is for a casual, 3D spatial puzzle game which would include some simplified geometric elements of architectural design. The user, their purpose for using the application, and their motivation while using the application are as follows:

- **User:** Casual Gamer
- **Purpose**: Playing a spatial puzzle game (where different configurations can net the player a different score based upon different categories). The rules used in the different puzzle levels can be applied to later levels
- **Motivation:** Solve puzzle, obtain a good final score

In this scenario the user, here referred to as the player, progress through a series of levels, each containing a set of shapes (objects) and room functions (strings). The player would go through a process of *new object instantiation*, *object manipulation*, and *object naming*. While moving through this process, the game will store in a list the actions they take in the form of rules. *New object instantiation*

would involve the player designation a shape from the provided list to be placed into the current level scene. *Object manipulation* would involve the process in which the player would determine the location and point of connection for the newly instantiated shape. *Object naming* would involve the player applying a room function from the provided list to the newly instantiated and manipulated shape.

Figure 2-10 Scenario 1 User-Action-Cycle
Source: Author

While the player goes through this action-cycle, the game saves their choices as *rules*. These rules can be later be used to provide the player with hints should they get stuck. The player will continue doing this until they have placed all provided shapes and assigned all of the room functions, whereby they will have finished the "puzzle" and be scored based upon a set of criteria. As the player progress from level to level, they would be provided a higher number of shapes and room functions.

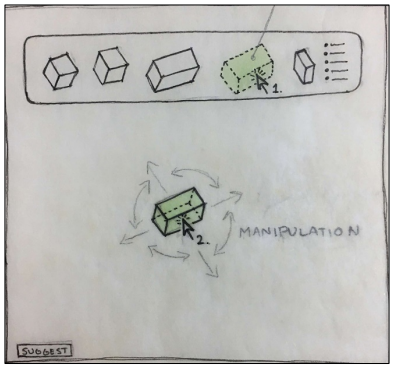


Figure 2-11 Scenario 1
Storyboard Panel 1
Source: Author

1. The player is provided with a set of shapes and a list of required functions (e.g. *"kitchen", "lobby"*, etc.

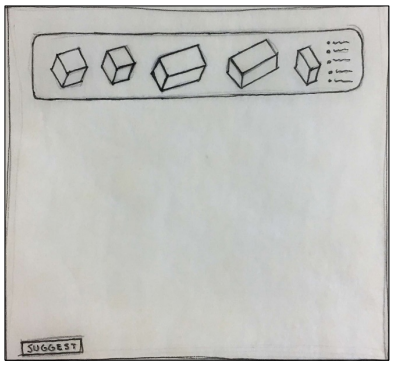


Figure 2-12 Scenario 1
Storyboard Panel 2
Source: Author

2. The player picks a shape to instantiate in the game space (the scene). They then choose its location and orientation.

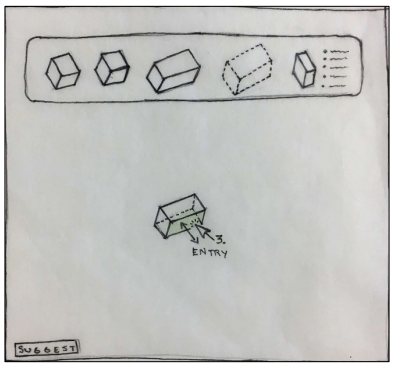


Figure 2-13 Scenario 1
Storyboard Panel 3
Source: Author

3. The player designates where the entry point is located on the shape.

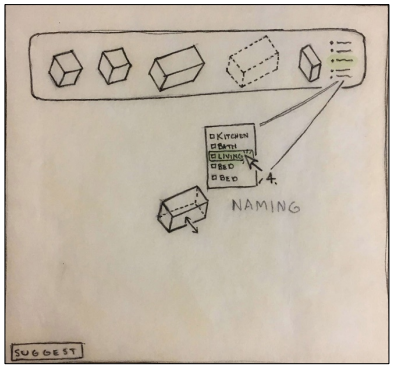


Figure 2-14 Scenario 1
Storyboard Panel 4
Source: Author

4. The player selects *"living"* to be the function assigned to the shape.

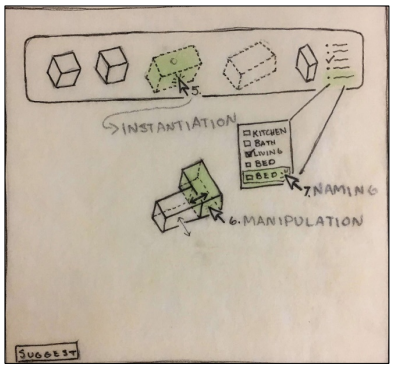


Figure 2-15 Scenario 1
Storyboard Panel 5
Source: Author

5. The player repeats the process seen in steps 2-3 to place/instantiate another shape into the scene. They then designate how this new shape attaches to the old space and then applies the label *"bedroom"* to the new shape.

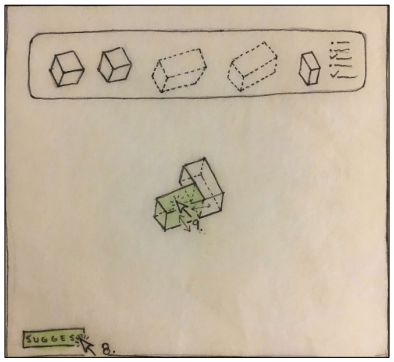


Figure 2-16 Scenario 1
Storyboard Panel 6
Source: Author

6. The player feels stuck, so since they have gone through a different level prior to this, they can click on the *"Suggest"* button to see the kind of spaces they have placed in the prior levels based off of either the *"living"* or *"bedroom"* labeled shapes that are in the present scene.

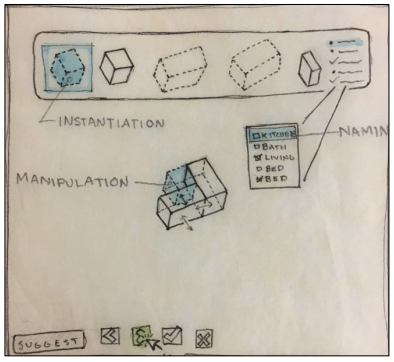


Figure 2-17 Scenario 1
Storyboard Panel 7
Source: Author

7. The game auto-selects a next shape, connection, and function. The player does not like this specific shape/connection/function combo so they click on the *"next"* arrow button to see a different suggestion.

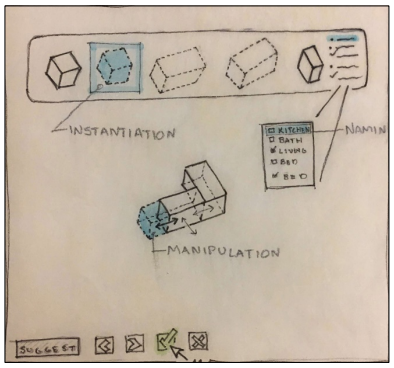


Figure 2-18 Scenario 1
Storyboard Panel 8
Source: Author

8. A new shape/connection/function combo is auto-selected by the game. The player decides that they like this one so they click on the *"OK"* button to confirm the rule.

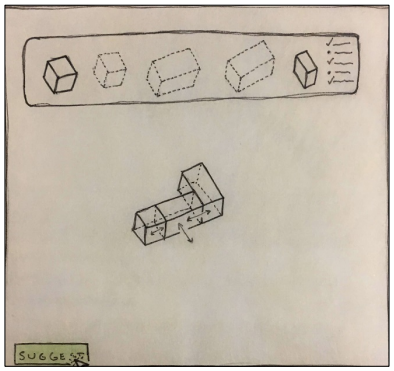


Figure 2-19 Scenario 1
Storyboard Panel 9
Source: Author

9. The player repeats the process for rule suggestion in steps 6-8 to continue getting suggestions based off of previous spaces.

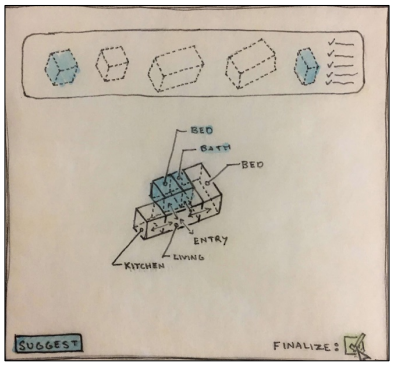


Figure 2-20 Scenario 1
Storyboard Panel 10
Source: Author

10. Once the player has placed all the shapes and assigned all the functions, they click on the "finalize" button to finish this level and obtain a score for their solution.

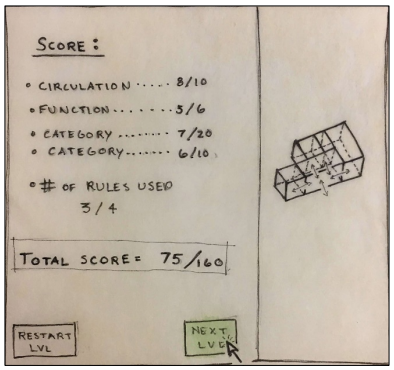


Figure 2-21 Scenario 1
Storyboard Panel 11
Source: Author

11. The score for the player's puzzle solution appears. The score is based on a list of criteria/categories plus how many of their own rules they have used. The player is happy with their overall score so they move onto the next level.

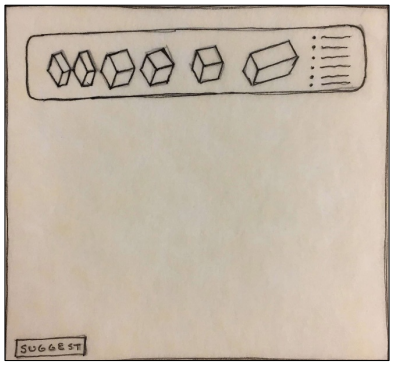


Figure 2-22 Scenario 1
Storyboard Panel 12
Source: Author

12. The player is taken to the next level where they are given a larger list of shapes and functions to work with

This scenario is very similar to what was being working on before 2018. However, it does not answer the question of where the shapes and room functions would come from or what would provide the basis for the scoring criteria. These questions would need addressing in any further development of this scenario.

2.4 Scenario 2: Modular System Visualization Tool

Scenario 2 attempts to think of an application that would functionw as a design tool, used in an architectural practice setting. The user, their purpose for using the application, and their motivation while using the application are as follows:

- **User:** Designer

- **Purpose**: To test out various compositions for a modular design system they are working on and to assist in communication those possibilities to their clients and team members.
- **Motivation:** Communication design variations.

Here the user would build 3D models for their modular units and use the proposed application to test out various configurations using those modular units and a set of placement constraints set by the user. The user would import the models for their modular units into the proposed application, *naming* them by assigning necessary information about each of the modular unit. After they had imported the units they wanted they would then define how each of the units could connect with one another, going through a process of n*ew object instantiation*, *manipulation*, and *connection*. *New object instantiation* involves the user designating one of their imported units they want to use. *Manipulation* involves the process in which the user decides where within the scene they want to place the unit they had just designated. *Connection* is where the user defines how the newly instantiated and manipulated unit connects to any adjacent units. The application would store the user's actions as rules, which could be used to create various design configurations for the user-defined modular system.

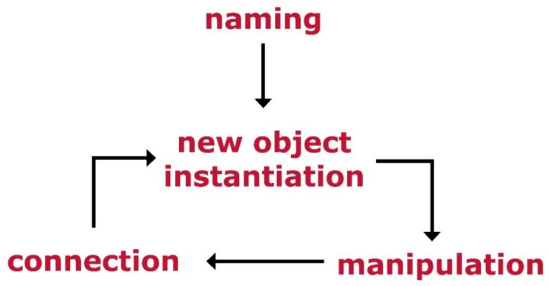


Figure 2-23 Scenario 2 User Action Cycle
Source: Author

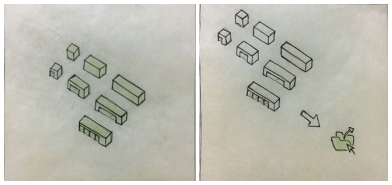


Figure 2-24 Scenario 2 Storyboard Panel 1
Source: Author

1. The user models their modular units they wish to use in a different program. They can then export those 3D models to be imported into *Unity* program.

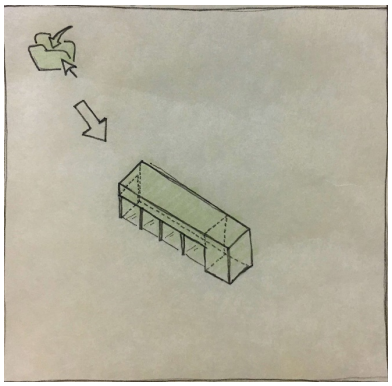


Figure 2-25 Scenario 2
Storyboard Panel 2
Source: Author

2. The user imports one of their 3D model units.

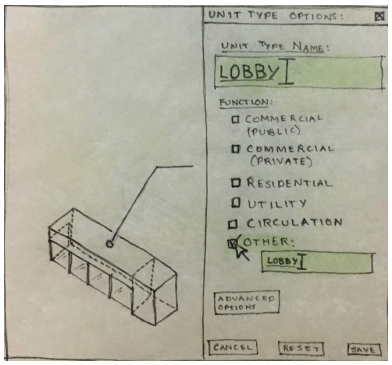


Figure 2-26 Scenario 2
Storyboard Panel 3
Source: Author

3. The user then provides some basic information about the unit type they have just imported, naming the unit type: *"lobby"* and since there is not a *"lobby"* or *"entry"* function listed, they select *"other"* and specify *"lobby"* in the text field.

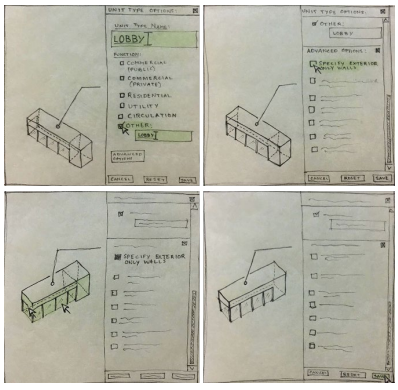


Figure 2-27 Scenario 2
Storyboard Panel 4
Source: Author

4. The user also specifies any other important information about the *"lobby"* unit type.

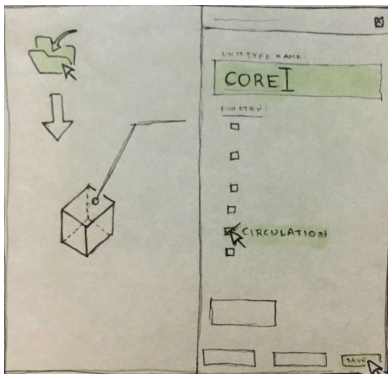


Figure 2-28 Scenario 2
Storyboard Panel 5
Source: Author

5. The user then imports another unit model type and sets its name to *"core"* and its function to *"circulation"*.

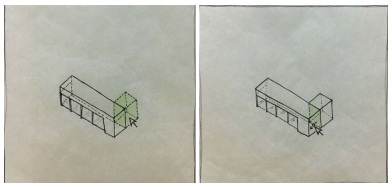


Figure 2-29 Scenario 2
Storyboard Panel 6
Source: Author

6. The user then places a *"core"* unit in the scene in relation to the initial *"lobby"* unit. They then specify how/where these units connect.

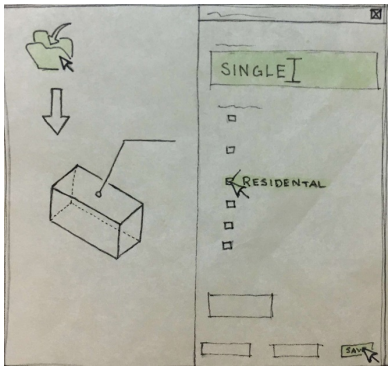


Figure 2-30 Scenario 2
Storyboard Panel 7
Source: Author

7. The user then imports the next unit model type and names it *"single"* and sets its function to *"residential"*.

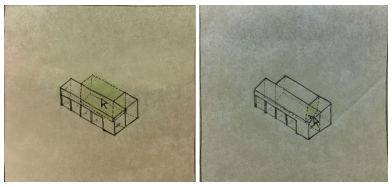


Figure 2-31 Scenario 2
Storyboard Panel 8
Source: Author

8. The user then places a *"single"* unit in the scene in relation to the initial *"core"* unit. They then specify how/where these units connect.

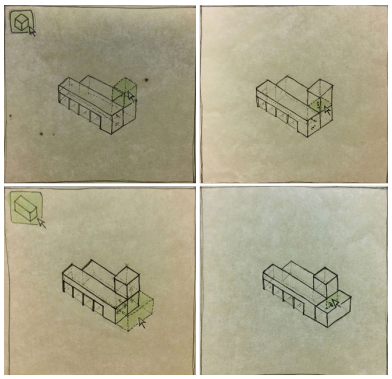


Figure 2-32 Scenario 2
Storyboard Panel 9
Source: Author

9. The user repeats this process to place an additional *"core"* and *"single"* unit in the model.

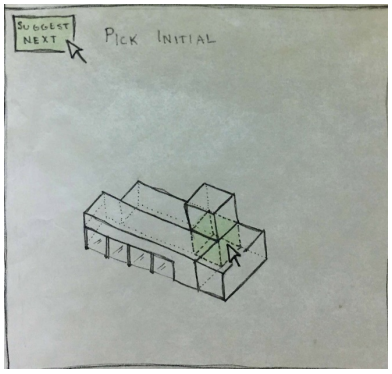


Figure 2-33 Scenario 2
Storyboard Panel 10
Source: Author

10. The user wants to test out some of the rules they have created so far, so they click the *"suggest next"* button and specify the *initial object* they want to base their next object off of.

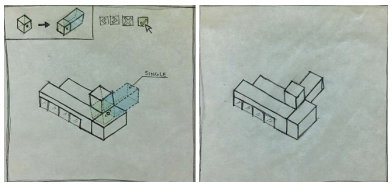


Figure 2-34 Scenario 2
Storyboard Panel 11
Source: Author

11. The game finds a suitable *next object*, the user finds it acceptable so they click the *"OK"* button to confirm this suggestion.

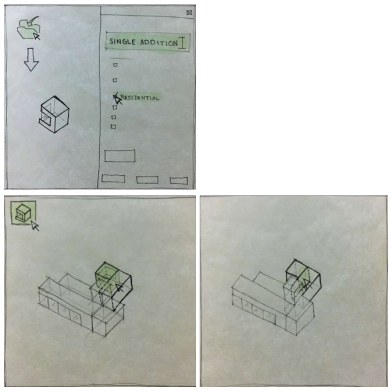


Figure 2-35 Scenario 2
Storyboard Panel 12
Source: Author

12. The user imports a new unity type they name *"single addition"* and give it the function *"residential"*. They then place it within the model space, specifying how/where it connects with the *"single"* unit type.

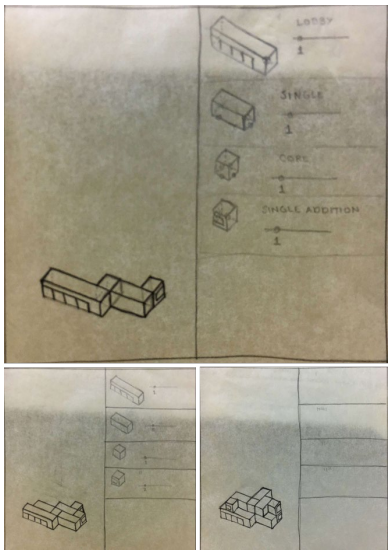


Figure 2-36 Scenario 2
Storyboard Panel 13
Source: Author

13. In a new model space, the user tests out overall designs they can create using the rules they defined above.

Towards the end of the storyboarding process for scenario 2, it became apparent that this scenario was too broad in scope and overly complicated, and it was decided that this scenario would not see any further development in this project.

2.5 Scenario 3: Precedent Game

Scenario 3 is a continuation of scenario 1 which seeks to answer the questions that arose upon reviewing said scenario regarding the basis for deciding on the shapes and room functions provided to the player and scoring criteria. Scenario 3 address that by using architectural precedents to determine the shapes and room functions in addition to the scoring criteria. The purpose and motivations for this scenario can be viewd from two different perspectives: there is the user's purpose and motivation and the developer/researcher's purpose and motivation. The user, their purpose for using the application, and their motivation while using the application are as follows:

- **User:** Student / Gamer
- **Purpose**: Learning about *precedents* through "Tetris-like" puzzles & *shape grammar*.
- **Motivation:**
    - Solve Puzzle and obtain a good final score.
    - Freedom of form making, ignoring final score.
    - Solving puzzles and sharing solutions with friends.

On the research/development end, our purpose and motivation are as follows:

- **Purpose:** Collect data on players' step-by-step process as they solve a presented spatial puzzle
- **Motivation:** Answer questions regarding commonly employed design rules.

In this scenario, a bit more work needed to be done to refine what specifically the purpose of the proposed application would be. In regards to the purpose of the application, it shifted in focus as it evolved from the work I had done prior to this project and scenario 1. While, originally I was planning on focusing on developing an

application to help people learn about shape grammar (see section 2.1) for this scenario I have opted to focus more on the education of precedents using some of the structure of shape grammar and design schema in the background, as there are already programs under development that focus on the direct application of shape grammar like GRAPE (a general parametric shape grammar interpreter) and CityEngine.[5] This would mean that the player would not directly interface with the concept of shape grammar nor use the rules they generate during gameplay necessarily. However, those rules would none the less be logged and gathered for research purposes, to see how players would progress through the puzzles, tying more closely into one of the questions asked at the beginning of this project: whether there are shape rules we as humans tend to use with higher frequency when designing, knowingly or not, and if so, what are these commonly used rules?

When looking into which architectural precedents to use a further examination of what aspects of the research done within the area of shape grammar to analyze particular architectural styles was done, with the goal of brainstorming ideas of how they might be able to be adapted to this video game format/presentation. These included a handful of Frank Lloyd Wright's prairie style houses, Siza's houses at Malagueira, Palladian villas, and Sir Christopher Wren's city churches.[6] During this examination two main ways for a designer to progress through architectural design arose:

1. Addition of new spaces adjacent to existing ones
2. Division of larger spaces into smaller ones

These two different processes became the two sub-scenarios for this scenario. Sub-sections 2.5.1 and 2.5.2 go into further detail for both of these sub-scenarios.

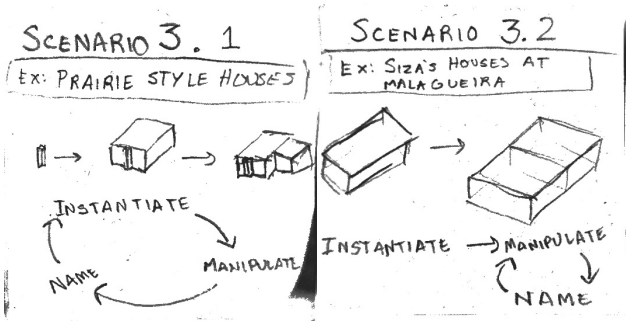


Figure 2-37 Scenario 3 Two Variations Brainstorming

Source: Author

2.5.1 Scenario 3.1: Addition

This scenario primarily draws heavily from work done by Koning and Eizenberg regarding their development of shape for Frank Lloyd Wright's prairie style houses. Addition refers to the process of adding additional shapes onto and around an initial shape similar in nature to the grammar and subsequent design schemata Koning and Eizenberg used to describe Frank Lloyd Wright's Prairie Style House. [7]

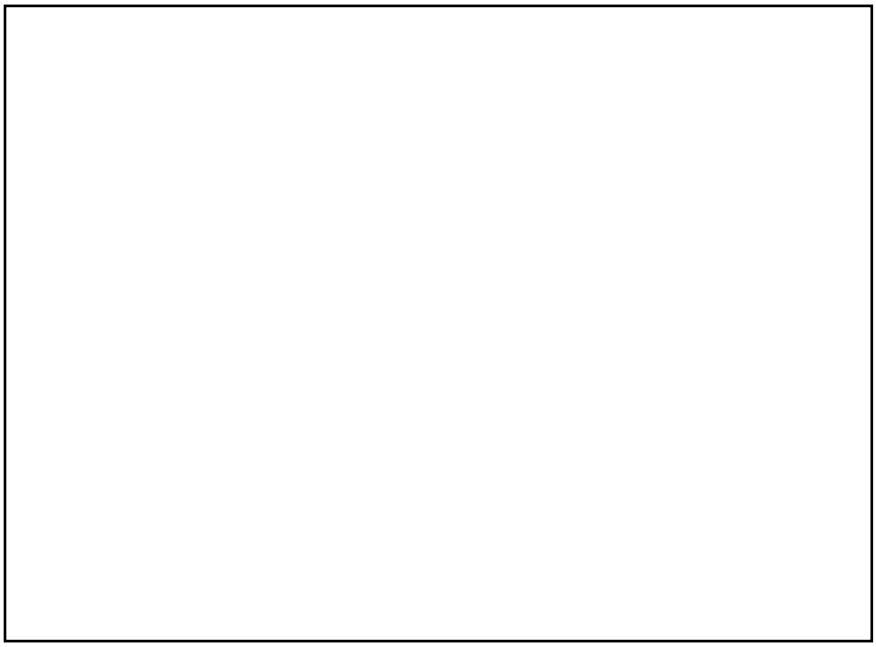


Figure 2-38 Frank Lloyd Wright's Prairie Style Houses Design Schemata
Source: Koning and Eizenberg

The "Addition" version of the precedent game is the most similar to the "Tetris-Like" Puzzle Game outlined in section 2.3 of the two versions for scenario 3. Both scenario 3.1 and scenario 1 share the same user action cycle. As a result, the storyboard for scenario 1 works for this scenario as well. The main difference between the two scenarios is that a selected architectural precedent would determine the shapes,

Figure 2-39 Scenario 3.1 User Action Cycle
Source: Author

labels, and scoring criteria in scenario 3.1 where they were just determined by the developer in scenario 1.

Scenario 3.1 is the scenario that was chosen for continued development as it was similar to the work I had already done before 2018, detailed in section 2.1 Preliminary Video Game Application Exploration. This scenario is also relatively straight forward, which should ease its transition from paper to implementation. More about the reasoning behind selecting this scenario is found in section 2.6.

2.5.2 Scenario 3.2: Division

This scenario takes its inspiration primarily from the shape grammar developed for Siza's houses at Malagueira, where a larger shape goes through a process of repeated division.[8] This process of division is also found to some extent several other shape grammars including grammars for the Palladian villa and Sir Cristopher Wren's city churches.[9]

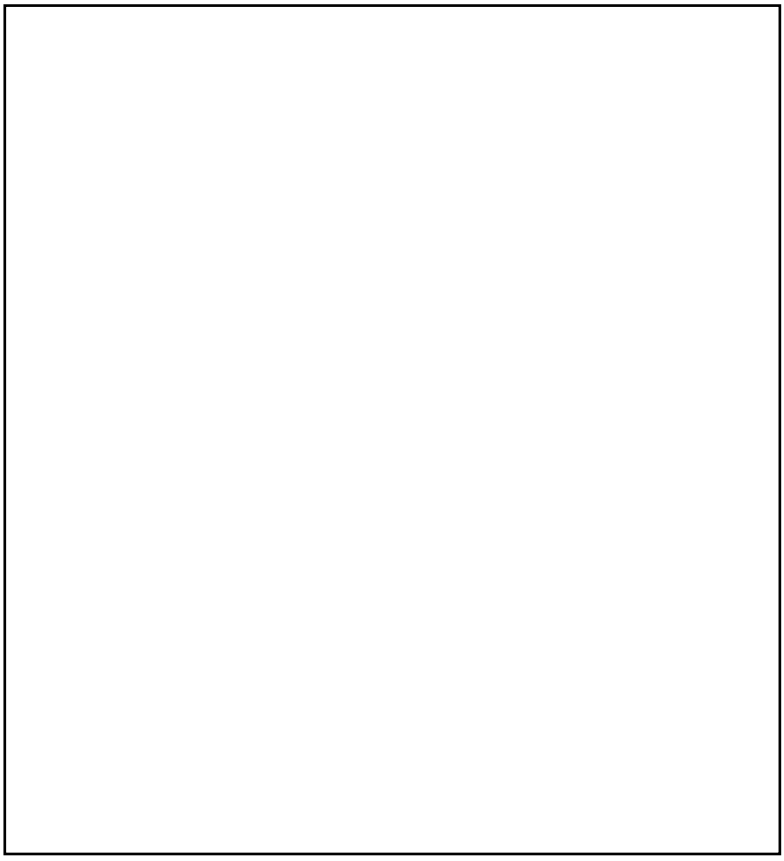


Figure 2-40 Siza's House at Malagueira

Source: Duarte

The "Division" version for the precedent game, while maintaining some of the same characteristics as scenario 1, the "Tetris-Like" Puzzle Game, would differ from scenario 1 in a significant way. Namely, that scenario 3.2 would already start with an initial object already instantiated within the game space/scene, and the player, instead of instantiating new objects repeatedly, would designate where an already existing game object would be divided. Within this process of dividing a larger object is replaced by two smaller objects, and the player would specify which object would maintain the initial object's room function and assign a new room function to the other.

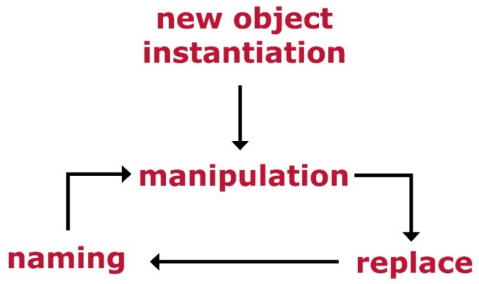


Figure 2-41 Scenario 3.2 User Action Cycle
Source: Author

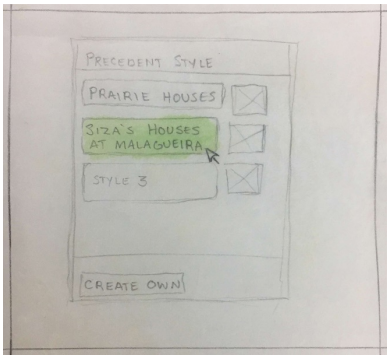


Figure 2-42 Scenario 3.2 Storyboard Panel 1
Source: Author

1. The player picks a precedent style from a provided library of precedents to work with (in this case, the player selects one of the Siza's Houses at Malagueira)

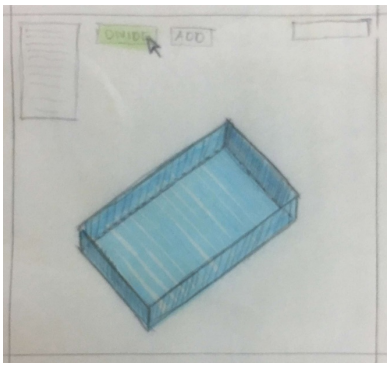


Figure 2-43 Scenario 3.2 Storyboard Panel 2
Source: Author

2. The player decides to divide the existing mass (opposed to adding on an additional mass), selecting the relevant button option.

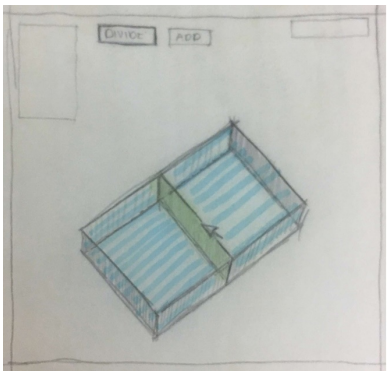


Figure 2-44 Scenario 3.2 Storyboard Panel 3
Source: Author

3. The player picks where they want this division to take place within the existing mass.

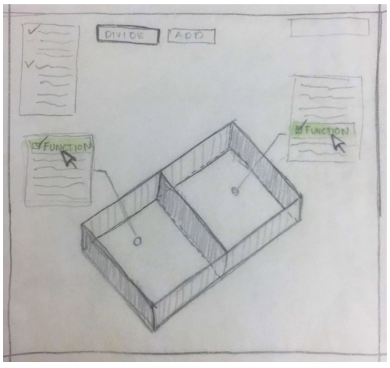


Figure 2-45 Scenario 3.2 Storyboard Panel 4
Source: Author

4. The player designates which functions go to the two newly instantiated massings, one being based off of the function of the initial mass, and the other being a new one selected by the player.

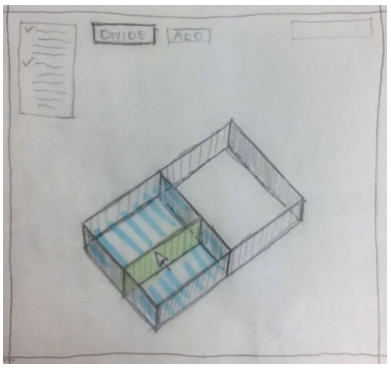


Figure 2-46 Scenario 3.2
Storyboard Panel 5
Source: Author

5. Still in "Division" mode, player decides which mass to divide next and locates where to place said division

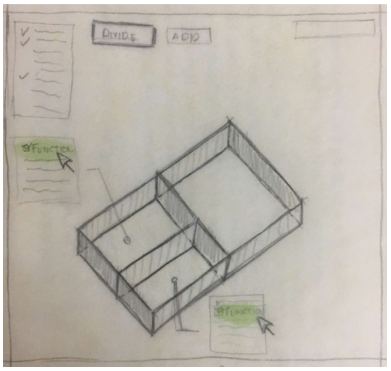


Figure 2-47 Scenario 3.2
Storyboard Panel 6
Source: Author

6. Player decides which mass retains the function of the initial and assigns a new function to the other

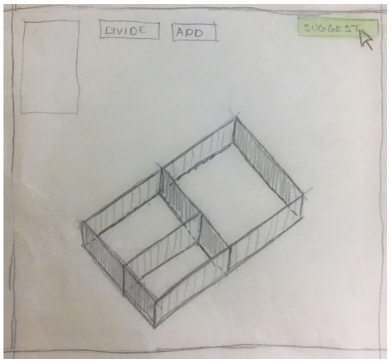


Figure 2-48 Scenario 3.2
Storyboard Panel 7
Source: Author

7. Player decides they want help so they click on the "Suggest" button

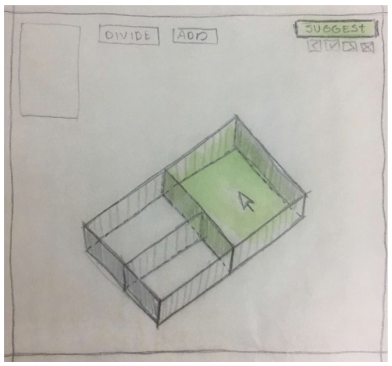


Figure 2-49 Scenario 3.2
Storyboard Panel 8
Source: Author

8. Player then selects a space within the game to suggest an option for using list of rules.

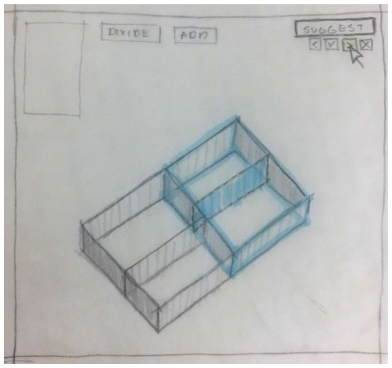


Figure 2-50 Scenario 3.2
Storyboard Panel 9
Source: Author

9. Game suggests a division for the player to use based on selected space and a corresponding rule.

While this scenario was intriguing, it was decided to be put on hold given the limited timeframe of this project and the work I had done prior using Kindergarten grammars. While this idea will not be further fleshed out within this project, it is nevertheless an interesting idea that may potentially find a use for being incorporated into the proposed game at a later date.

2.6 Scenario Selection Summary

After reviewing the literature about shape grammar and its various applications within the field of design and design education, several different scenarios of potential digital applications where developed to explore potential project ideas and directions. The first three scenarios were as follows:

- Scenario 1: "Tetris-Like" Puzzle Game
- Scenario 2: Modular System Visualization Tool
- Scenario 3: Precedent Game

With the third scenario being divided up into two different scenarios:

- Scenario 3.1: Addition
- Scenario 3.2: Division

Scenario 1 detailed a video game in which a player is presented with a set of 3D shapes and a list of room functions which they would need to combine in order to "solve" a game level. After the player had "solved" the level, they would be provided with a score before being taken to another level where they are provided a longer set of 3D shapes and room functions while this scenario showed a great deal of promise in terms of feasibility based on the experimentation done using *Unity* before this project. However, it also had some significant drawbacks. Namely, what/how a developer would select what shapes and room functions a player is given in each level, and what would serve as the basis for the scoring system is unclear?

Scenario 2 looked to approach the problem and potential solution in terms of architectural practice and modular design. This scenario did not see too much development as early on it was discovered to be outside of the scope of what I wanted to do and too similar to what was already out there without really bringing much new to the table.

Scenario 3 was a continuation of scenario 1 that sought to answer its major drawback of not basing the levels on anything concrete by using architectural precedents as the backbone for puzzle levels. When deciding how to go from architectural precedent to 3D puzzles, more research was done into how researchers have developed languages to explain and understand various architectural precedents and their styles using shape grammar. Based on this research, scenario 3 was split up into two different scenarios. Scenario 3.1 details an application/game which would look virtually identical to the one detailed in scenario 1, but each puzzle level would be based on a particular precedent. This scenario is optimal in terms of feasibility but might run into some trouble when it comes to architectural precedents which are not easily understood and were not arrived at through a process of

addition. Scenario 3.2 sought to address precedents that have languages which heavily utilize the divisions of larger spaces into smaller ones. This scenario, therefore, details an application/game in which a player would go through a process of division to arrive at a solution.

After looking at the scenarios discussed in this chapter, Scenario 3.1: Precedent Game – Addition, was selected as the best choice moving forward. Not only did Scenario 3 better address the problem statement outlined in the first chapter, but Scenario 3.1, in particular, seemed a better fit for this type of early experimentation in developing the type of game outlined in Scenario 3 based upon the preliminary research and experimentation I had done before the start of this project detailed in section 2.1. The remaining chapters will detail the steps that have made towards the implementation of 3.2, and what the next steps in its development might be post-dissertation.

---

[1] Kari Kuutti, "Work Processes: Scenarios as a Preliminary Vocabulary," in *Scenario-based design : envisioning work and technology in systems development* (New York: New York : Wiley, 1995).

[2] Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[3] Stiny, "Kindergarten grammars: designing with Froebel's building gifts."

[4] Example Precedent Images Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[5] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."

[6] Buelinckx, "Wren's Language of City Church Designs: A Formal Generative Classification."; Duarte, "Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira."; Koning and Eizenberg, "The Language

of the Prairie: Frank Lloyd Wright's Prairie Houses."; Stiny and Mitchell, "The Palladian Grammar."

[7] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[8] Duarte, "Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira."

[9] Buelinckx, "Wren's Language of City Church Designs: A Formal Generative Classification."; Stiny and Mitchell, "The Palladian Grammar."

Chapter 3: Abstraction Methodology

The scenario selected in Chapter 2, Scenario 3, describes a 3D puzzle game which would use various architectural precedents as the basis for the shapes and labels – the puzzle pieces – provided to a player at the beginning of each puzzle level. In order to create the puzzle pieces for each level, the precedent assigned to the level goes through an abstraction process. This process involves breaking down the precedent's overall layout into its basic massings, accounting for both formal and functional aspects of the design. These massings serve as the shape component of the puzzle piece game object and will be referred to throughout the work as either *shapes, game object, puzzle pieces* or *puzzle piece game objects* depending on the context. *Shape* is used within the context of architectural design, shape grammar, and the model components of each *puzzle piece. Game object, puzzle piece,* and *puzzle piece game object* are interchangeable and used when discussing the development of the game using the *Unity* game engine. Labels are assigned to the *puzzle pieces* and refer to the room names/functions belonging to the precedent's room massing which correspond to the *puzzle piece's shape* components. These labels both provide and account for additional information relevant to the design within the grammar; in this case, the information is referring to functional arrangements and Space Syntax of the precedent.[1]

Section 3.1 will detail the process for selecting the two precedents used throughout this project: The Martin (Barton) House by Frank Lloyd Wright and the Villa Savoye by Le Corbusier. Sections 3.2 - 3.4 detail the abstraction process for both precedents, outlining the iterations each one underwent and what changes occurred between each along with the reasoning behind these changes. Section 3.5 summarizes some of the initial findings specific to the outcome of this abstraction process as it relates to shapes. The current implementation of this project only looks at one of the precedent levels of the two precedents that were selected for abstraction: the Martin (Barton) House by Frank Lloyd Wright.

3.1 Precedent Selection Overview

While the idea behind scenario 3 is that the precedents would belong to a variety of architectural typologies that could span a variety of scales, it was decided very early on that the alpha build which comprises outcome this project would focus on single-family houses. This decision was made in order to limit to the scope of the

project to something which achievable within the D.Arch timeframe. Because scenario 3.1 (described in Section 2.5.1) took some of its inspiration from the paper "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses" by H. Koning and J. Eizenberg which looked at Frank Lloyd Wright's houses, the houses discussed within that paper were a good place to start looking for precedents to use.[2] Two other precedents were also selected that could contrast with the prairie style house, with the goal of ensuring the overall abstraction process could take into account a greater variety of designs.

The initial handful of possible precedents I looked at were selected in large part based on if I had easy access to information about them and the two houses, I already had access to 3D models of. Those initial precedents where the following:

- The Martin House (Barton House) by Frank Lloyd Wright
- The Little House by Frank Lloyd Wright
- The Roberts House by Frank Lloyd Wright
- The Willets House by Frank Lloyd Wright
- The Koshino House by Tadao Ando
- The Villa Savoye by Le Corbusier

This list was further reducted to the following three by selecting a single Frank Lloyd Wright house to work with:

- The Martin House (Barton House) by Frank Lloyd Wright
- The Koshino House by Tadao Ando
- The Villa Savoye by Le Corbusier

The initial abstractions for each of these precedents are detailed in sections 3.1.1 through 3.1.3 .

After initial abstractions for each of the potential precedents, two precedents were selected which would contrast each other to allow for the exploration of as many different types of problems that would likely arise in abstraction of additional precedents should the game seek continued development after the conclusion of this dissertation project. The following two precedents where the ones selected:

- Martin House (Barton House) by Frank Lloyd Wright

- The Villa Savoye by Le Corbusier

Martin House (Barton House) by Frank Lloyd Wright was selected due to its reasonably simple geometric components and a floorplan that contains enough complexity to allow for easy creation of design variation. The Villa Savoye by Le Corbusier was selected due to its use of more complex and unique shapes that resisted abstraction to solely rectilinear geometries (a contrast to the Martin House).

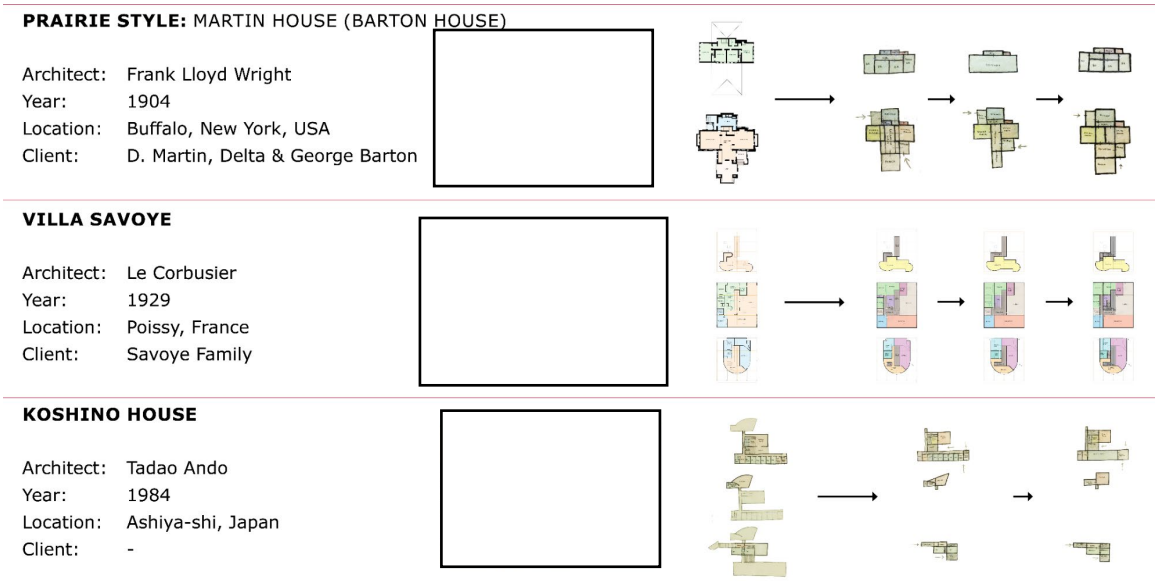


Figure 3-1 Overview of Preliminary Abstractions

Source: Author with Building Images from Wright, Sbriglio, & Andō[3]

3.1.1 Frank Lloyd Wright's Prairie Style Houses

Frank Lloyd Wright was a modern American architect operating in the early twentieth century. He is famous for is "organic" style of architecture and is much known for developing the Prairie Style in the early 1900s. The Prairie Style is known for its low-pitched roofs with large overhanging eves giving it a distinctly horizontal element inspired from its midwestern backdrop. The Prairie Style is also known for its distinct t-shaped floor plans with public and service spaces on the ground floor (or lower floors in the case of three-story houses) and private spaces on a smaller second or third floor.[4] Two criteria determined the houses of this style this project looked at as potential precedents to be implemented:

1. Was it abstracted in Koning & Eizenberg's paper cited throughout this dissertation?
2. Was there access to labeled floorplans for the precedent? (these had to be translated from the original German).

The four precedents from Frank Lloyd Wright's body of work examined by Koning and Eizenberg that met these criteria are the following:

- Martin House built in 1904
- Little House built in 1902
- Roberts House built in 1908
- Willet's House built in 1902[5]

The below are floor plans, their basic abstractions from Koning and Eizenberg's paper, plus an initial abstraction for the top floors for each of these houses.[6]

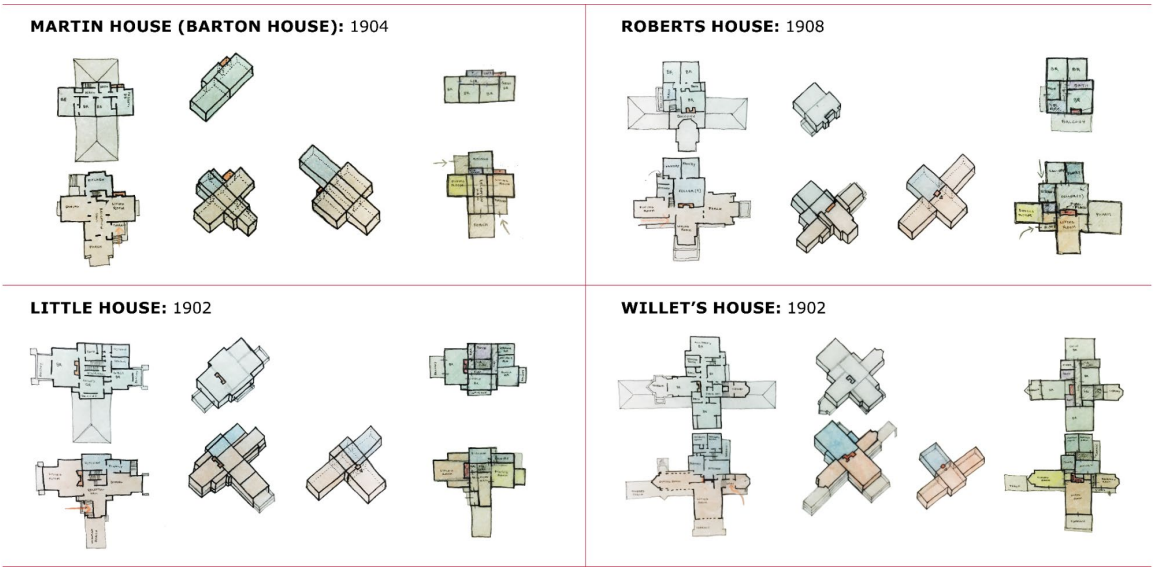


Figure 3-2 Preliminary Abstractions of Selected Prairie Style Homes

Source: Author

The Martin House (Barton House) was selected from these four houses to be look at further as it was not missing a third floorplan like the other three, and its floorplan has an overall internal composition that aligned within itself more than the

other three. These aspects of the Martin House's design eased the transition from the work done by Koning and Eizenberg to this project.

3.1.2 Martin House (Barton House)

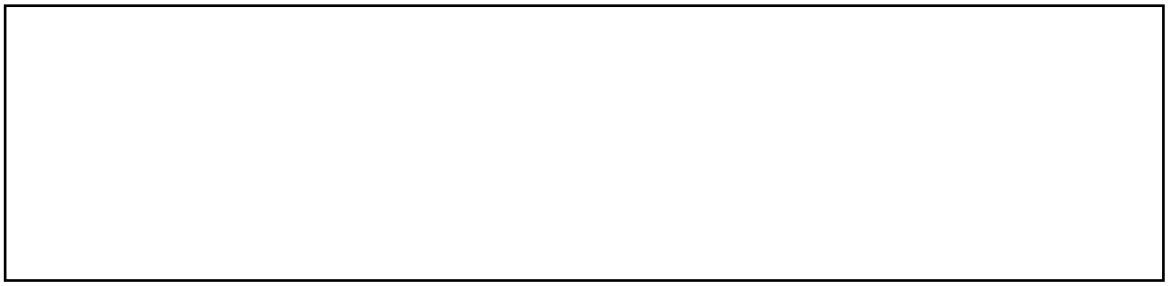


Figure 3-3 Images of the Martin House Complex and the Barton House[7]
Source: Frank Lloyd Wright

The Martin House is a building complex designed by Frank Lloyd Wright in 1904 in Buffalo, New York for Darwin D. Martin and Delta and George Barton.[8] The Martin House is a building complex, and the specific home I am looking at within the complex is called the Barton House (its location in relation to the rest of the complex is shown in Figure 3-3). It is worth noting that this portion of the Martin House complex is referred to as both the Martin House and the Barton House throughout this paper.

A number of different abstractions were generated for the Barton House before I felt confident in transitioning to physical and digital 3D models/mockups. The three initial abstractions for the Martin House are shown in Figure 3-5. It was during the creation of these three abstracted versions of the Barton House that some of the things to be look for in the process of abstraction were defined. These would then be used when the other precedents were initially abstracted for this portion of the project. These ideas include the following:

- The importance of finding consistent/uniform sizes and shapes throughout the design.
- The usefulness of complex shapes to help break down more complex room shapes.
- The indication of points of circulation connection between two rooms to help further articulate the design of the precedent.

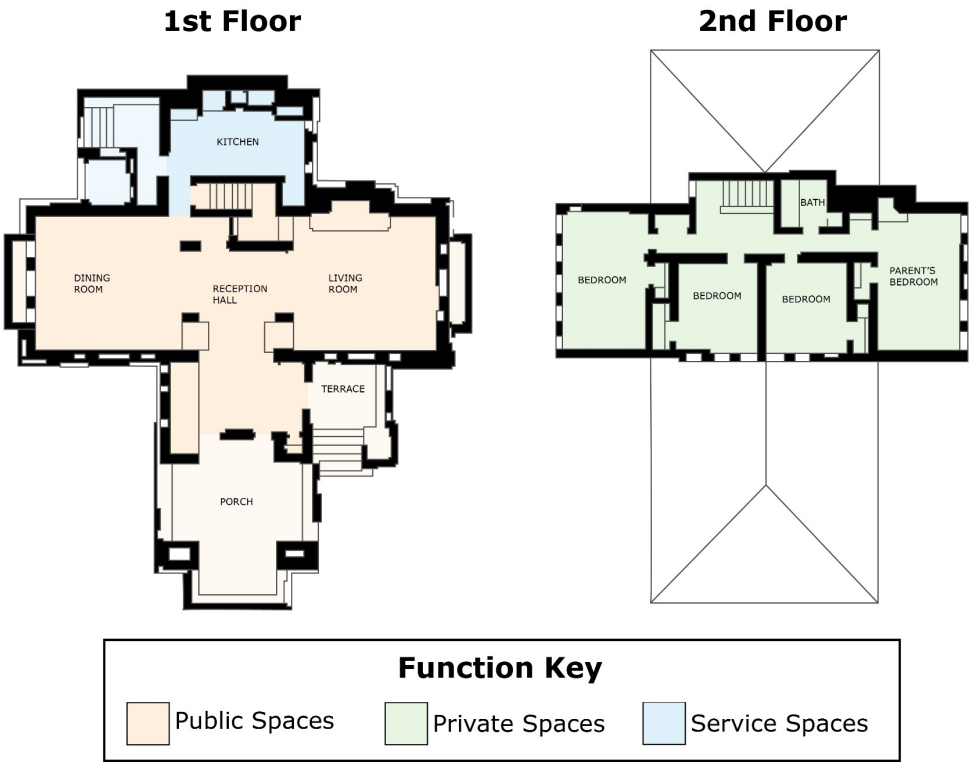


Figure 3-4 Martin House (Barton House) Floorplan by Wright [9]

Source: Author

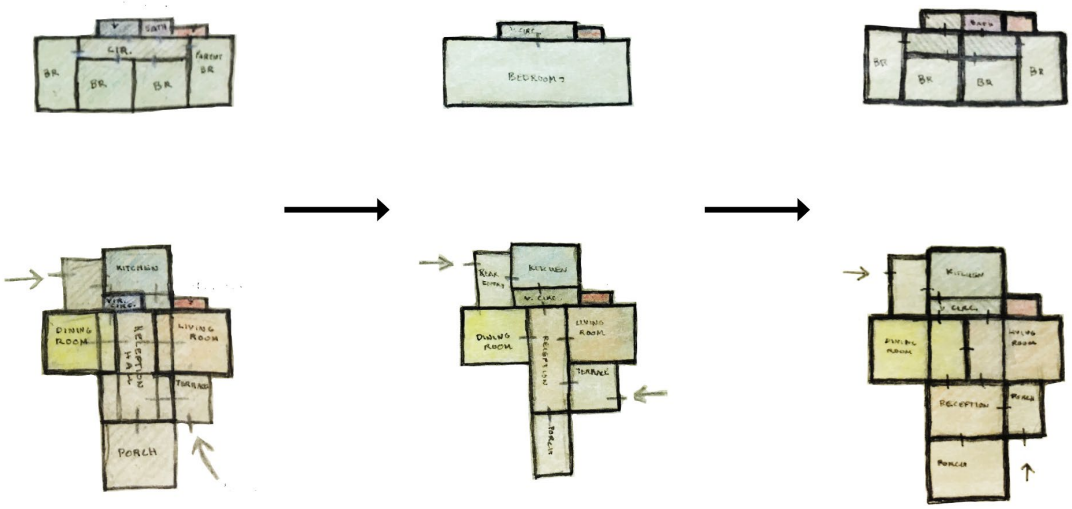


Figure 3-5 Initial Abstractions for the Martin House (Barton House)

Source: Author

3.1.3 Koshino House

Figure 3-6 Images of the Koshino House

Source: Tadao Ando[10]

Tadao Ando design the Koshino House in Ashiya, Huogo, Japan in 1979 with an addition added to the building in 1983.[11]

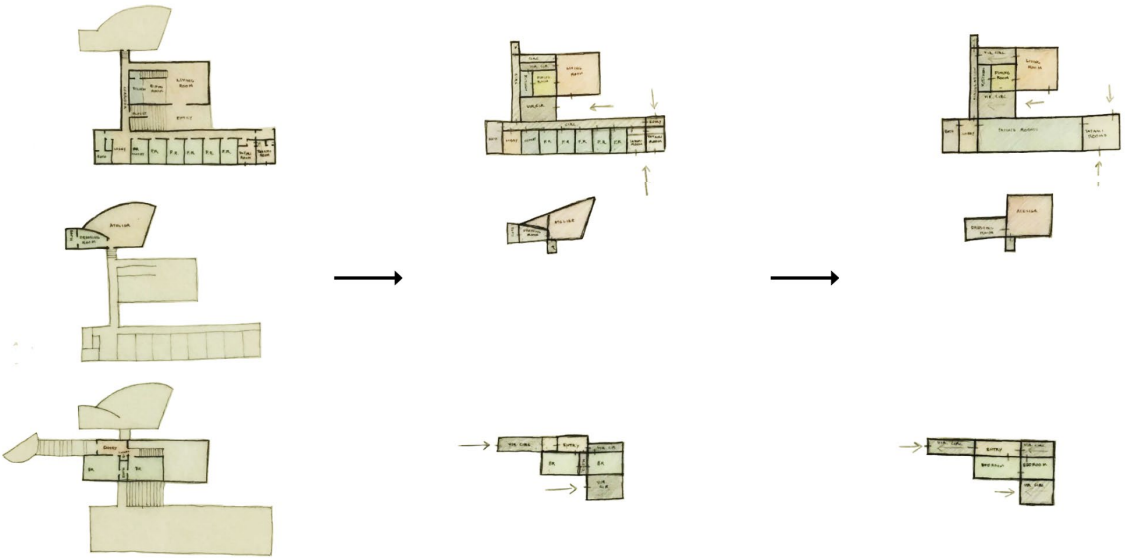


Figure 3-7 Koshino House Floor Plan & Preliminary Layout Abstractions[12]

Source: Author

While two initial versions of an abstracted layout were created, this precedent was not further explored due to a concern that the number of rooms might be too confusing for a player when they are putting the house back together. Despite the fact that I feel like this would not end up being a hurdle too large to get over based upon some of the usability issues and solutions discussed in later chapters of this project, its abstraction in preperation for implementation was put on hold as it only contains one unique shape and does not provide enough of a contract to the Martin House, especially when compared to the Villa Savoye which is looked at in the next

section. However, the Koshino House would still be a useful precedent to consider in early level development later on.

3.1.4 Villa Savoye

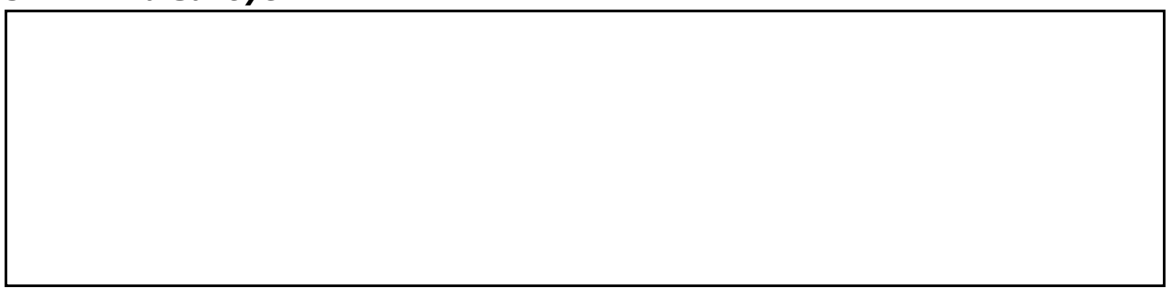


Figure 3-8 Images of the Villa Savoye[13]

Source:"Le Corbusier: The Villa Savoye" by Jacques Sbrigio

The Villa Savoye was design by Le Corbusier and built in 1929 in Poissy, France for the Savoye Family. It acts as a synthesis of a lot of the work Corbusier had been doing throughout the 1920s and his hunt for creating white "Purist villas." It is a design which played with the contrast of two ideas: *"the machine for living in"* and *"the machine for feeling."*[14]

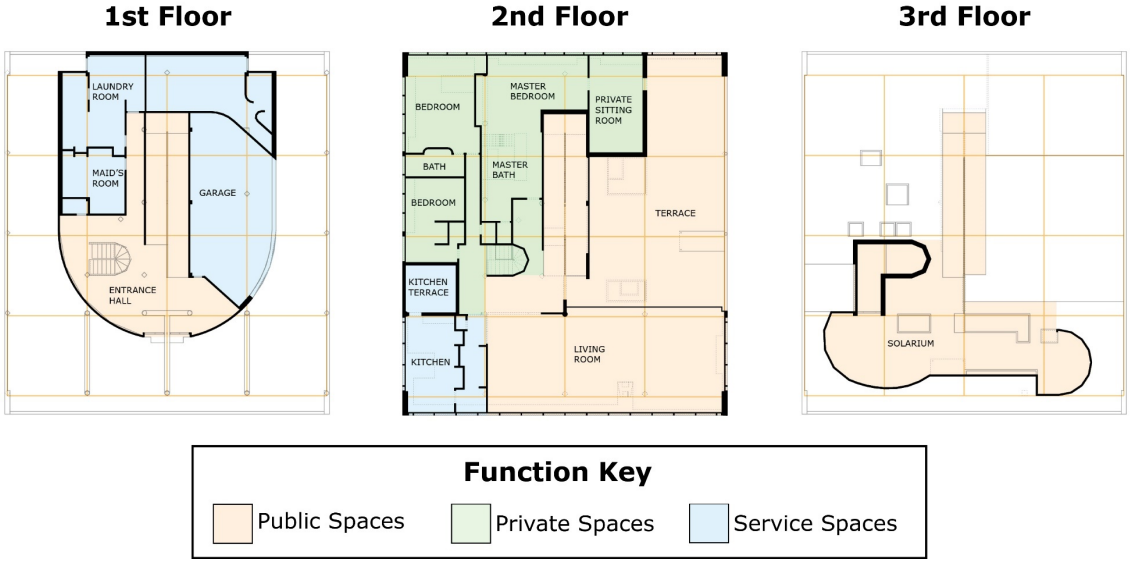


Figure 3-9 Villa Savoye Floor Plan[15]

Source: Author

Relevant to this project is that the Villa Savoye contains several wonky and curved shapes which contrast with the more rectilinear shapes typical in architectural design. Initially, it was hypothesized that this would make a 3D spatial puzzle based

on the Villa Savoye easier to solve than the more uniform Martin House. However, this hypothesis would end up being proven extremely wrong, as even the current iteration of the Villa Savoye level is not an easy one by any means. The process for abstracting the layout for the Villa Savoye also started differently than the process of creating initial abstractions for all of the other precedents, as the Villa Savoye started Rhino and never transitioned from a pen and paper sketched iteration. Because of this, the initial abstraction for the Villa Savoye is the same as the first iteration gone over in section 3.4.1 therefore, only the floor plans for the Villa Savoye are included in this section.

3.2 Layout Abstraction Process Overview

Both the Martin House and the Villa Savoye went through several different iterations of abstraction. Changes made between each iteration were done to help with solve player usability problems when it comes to both arriving at the original solution and the creation of design variations. Both the Martin House and Villa Savoye abstractions where worked on in parallel, with each iteration learning from the shortcomings of the previous. Most of the differences between each of the iterations have to do with how the layouts where divided into their different shapes.

The first iteration for both was an almost, if not, identical layout to the initial abstractions detailed in the previous section. Those abstracted layouts where taken and a physical (Martin House) or digital (Villa Savoye) model was made for testing usability. Iteration two removed some of the more complex elements from the first, removing compound shapes, replacing them with complex shapes and reducing the overall number of individual shapes wherever possible. In iteration three compound shapes were added back in to replace complex and unique shapes wherever possible, though the player defined grouping of these compound shapes was left out. Players were also given labeled shapes, as opposed to having to label them themselves as they did in iterations one and two. The fourth and final iteration looked to double down on the use of compound shapes in place of complex shapes wherever possible and the fourth iteration of the Villa Savoye specifically also addressed some problems that had arisen due to rounding errors compounded from the previous three abstracted iterations. Figure 3-10 details the fundamental differences between these iterations.

Rhino was used to create all of the digital models used in this abstraction process with the last iteration's individual model volumes being transferred into Unity for the development of their respective precedent puzzle level. Section 5.2.3 details the process of exporting the 3D models for each shape from Rhino to *Unity*.

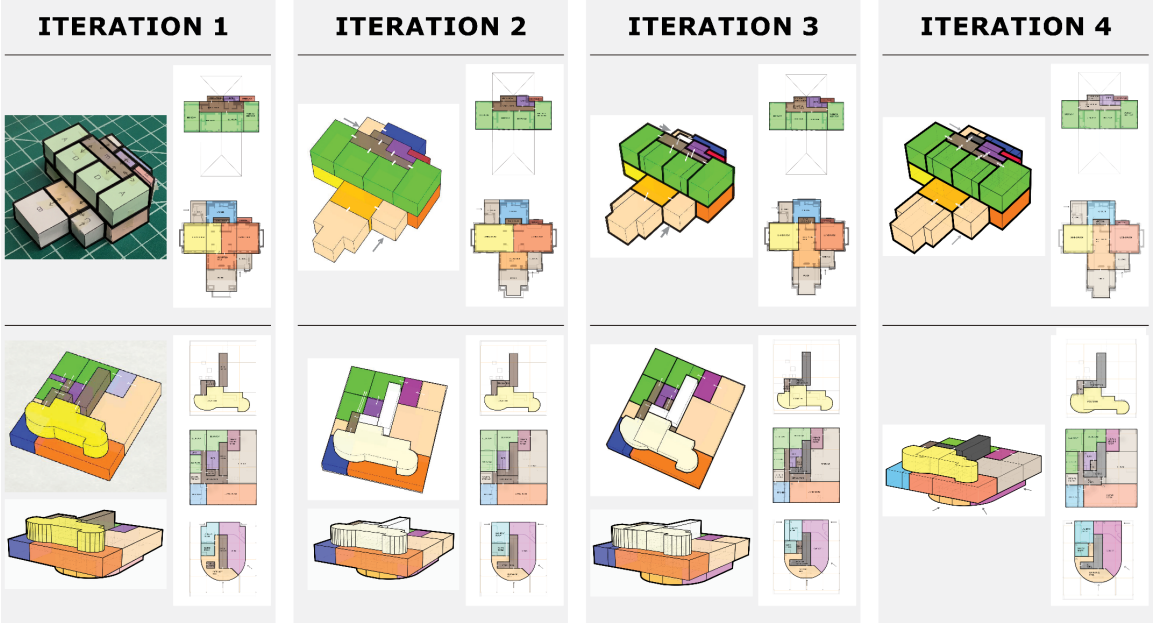


Figure 3-10 Overview of Abstraction Iteration

Source: Author

There are a few key things to keep in mind when abstracting that make not only solving each puzzle slightly more natural, but also assist in the possibility of having the player generate their own design alternatives that were discovered during this iterative process. These findings can be boiled down to two key goals:

- try and maintain some uniformity in dimension
- try and have as many edge conditions of both overall shapes and their sub-shapes align.

These ideas will be explored and expressed throughout the document but are worth mentioning here.

3.3 Abstraction: Martin (Barton) House

The Martin House (Barton House) abstractions have had the most extensive testing done using its 3D mockup model in Rhino, and has also served as the focal point when testing any adjustments to the abstraction process that do not specifically look at curved shapes. Figure 3-12 shows the transition from each of the four iterations to the next. The overall abstraction process for the Martin House proved to be reasonably straight forward, probably due in large part to the organic nature of the Prairie style, which lends itself well to this way of thinking spatial composition, where the design's overall composition is primarily developed through the addition of simle geometric shapes to each other in an iterative process.[16]

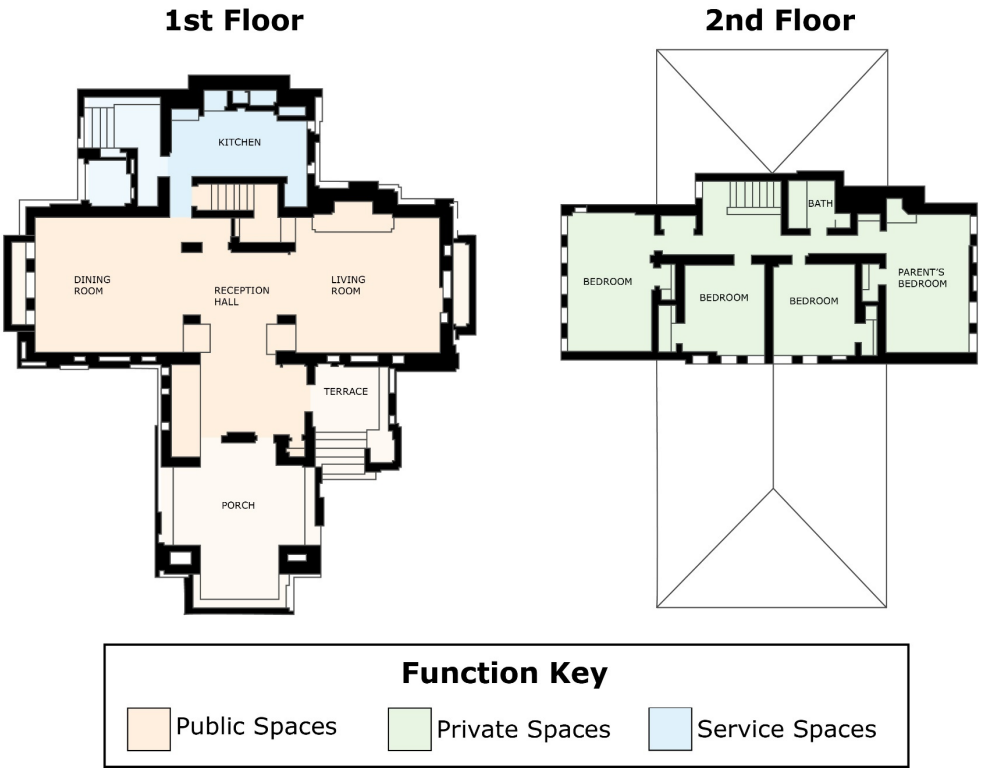


Figure 3-11 Martin House (Barton House) Floorplan by Wright

Source Author

Each iteration contains an abstracted floorplan and a functional layout bubble diagram that is a result of that abstraction. The bubble diagram is included because one of the scoring criteria proposed is accuracy of functional connections, as it is not only formal accuracy that is important to these designs.

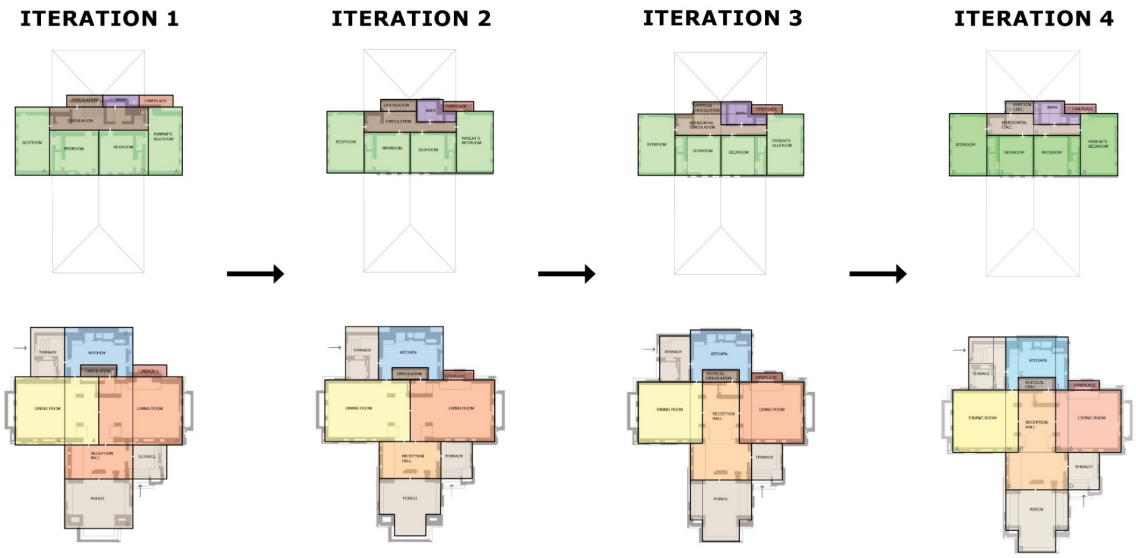


Figure 3-12 Martin House Abstraction Iterations

Source: Author

3.3.1 Iteration 1

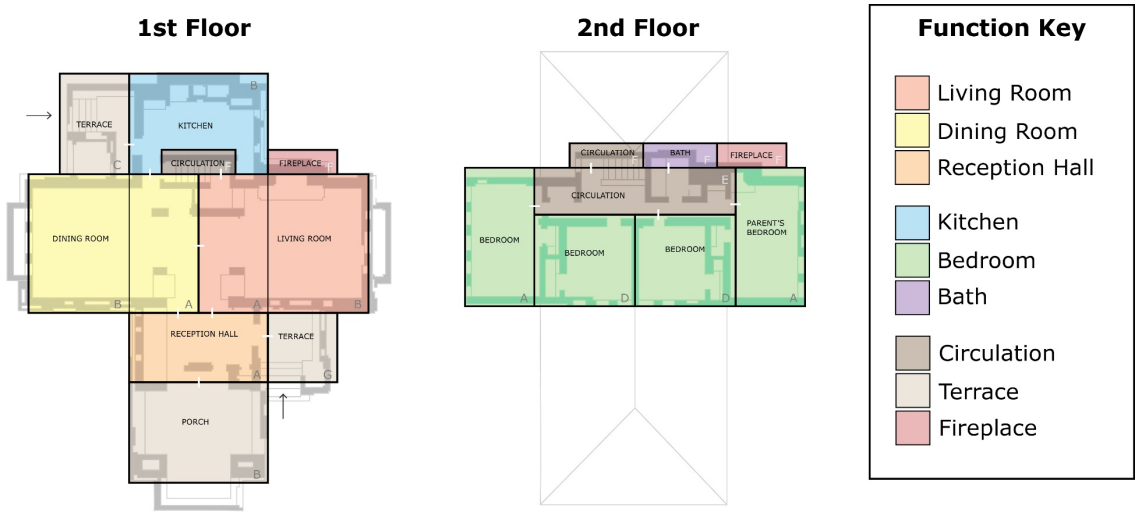


Figure 3-13 Martin House Iteration 1 - Abstracted Floorplan

Source: Author

The first iteration for an abstracted layout of the Martin House was a direct translation from a traced copy of the original floorplan. This iteration is also the only iteration to have been initially done using analog mediums. This abstraction contains 19 shapes in 7 different sizes, 14 different rooms (with 4 of them comprising of a compound shape made of 2 to 3 individual shapes).

In this iteration of the Martin House level, the player is provided with a set of shapes (see image below) and a list of rooms (labels to be applied to the shapes like "living room" or "Bedroom"). The player would then go through an iterative process of *new object instantiation, manipulation, naming,* and *defining connections*. *New object instanition* involves the player placing a new object into their scene. *Manipulation* involves the player moving it into place (and grouping it with another shape already within the scene if necessary). *Naming* is when the player applying a room function to it from the list provided. Finally, *define connection* is when the player designates how the newly instantiated shape would connect to shapes adjacent to it if necessary. The player would then repeat this cycle till they have exhausted both the list of shapes and room functions. This user action cycle is virtually identical to the user action cycle for scenario 1 (see section

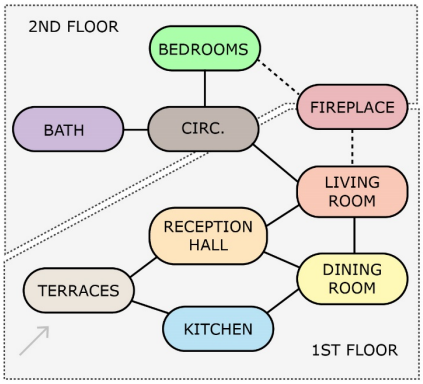


Figure 3-15 Martin House Iteration 1 – Function Bubble Diagram
Source: Author

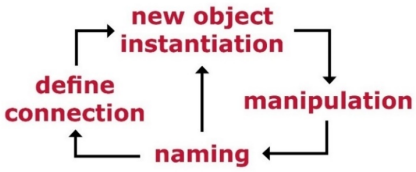


Figure 3-14 Martin House Iteration 1 – User Action Cycle

Because this iteration saw the first 3D models for any of the abstractions looked at for this project, it served as the preliminary means of testing how easy it was to reach the original solution (the one which prefectly matched the precedent) as well as the ease in which the player could arrive at design variations which made sense. A simplified version of this test that only look at the process of arriving at the original solution was repeated for every iteration of both the Martin House and the Villa Savoye.

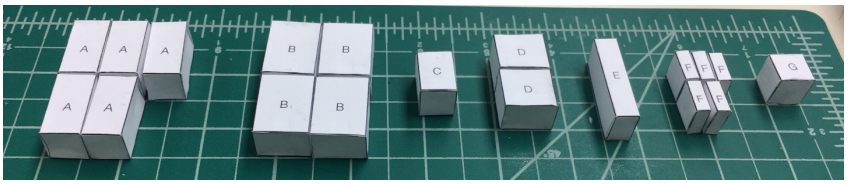


Figure 3-16 Martin House Iteration 1 – Shapes Provided to Player
Source: Author

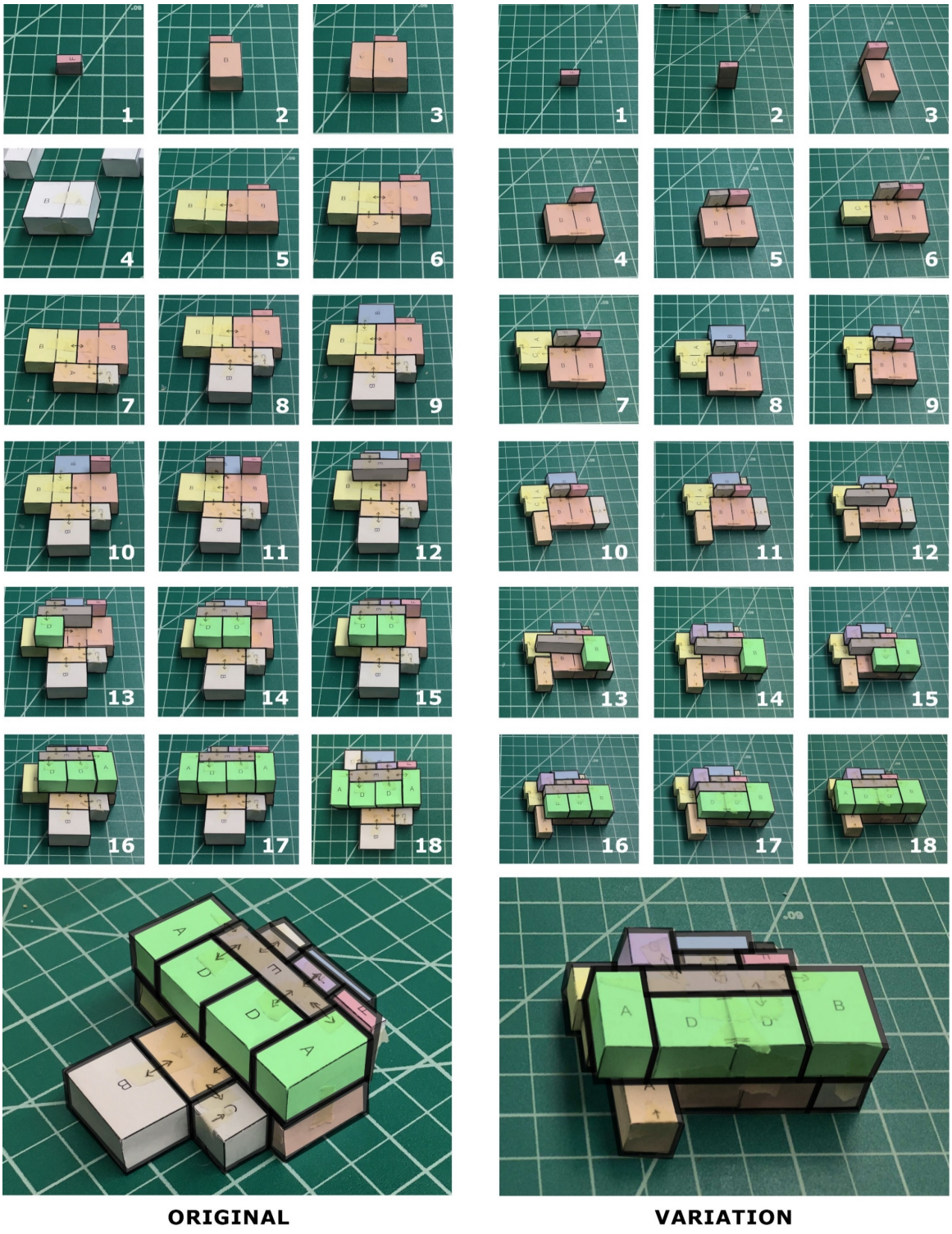


Figure 3-17 Martin House Iteration 1 - Original Solution & Variation

Source: Author

This iteration brought into light a few issues which required addressing in future iterations. One of the problem was that spaces looked like they should be align between the two floorplans in the drawings, but when translated into 3D space it was discovered that they did not, as was very much the case for the circulation which connected the first and second floors. Initially, it appeared that their footprints perfectly aligned, one on top of the other (like the image on the top), but instead they are slightly offset (like the bottom image). This problem also spoke to the fact that working primarily from trace paper is not a very precise means of figuring out dimensions, which can end up making a significant difference when small enough deviations start adding up and would ultimately prompt the transition into Rhino for all later iterations (including all the Villa Savoye iterations). Another key issue that arose that was specific to the circulation was the need to address overlapping shapes that occurred within the same floor. It quickly became apparent that in addition to simple rectangular shapes, complex shapes would need to be used in order to prevent shapes needing to be overlapped/nested within one another.

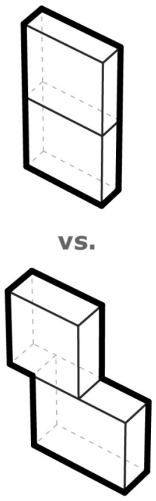


Figure 3-18 Stacking Shapes Iteration 1 (top) vs. Iteration 2 (Bottom)
Source: Author

Another thing that became apparent was that it would become essential to try and preserve uniform dimensions and shapes wherever possible. This uniformity would allow for more shapes to be able to be combined in different ways without resulting in weird overall shapes and gaps where it becomes evident that something is very wrong as shapes essentially stop lining up with one another. This idea is core to all other iterations of the Martin House.

3.3.2 Iteration 2

For the second iteration of the Martin House, the abstracted floorplan and modeling were all done digitally with Rhino. By necessity, this meant that some adjustments to the first iteration were needed. Part of this transition meant coming to grips with the fact that some of the shapes that appeared to be the same size on trace paper ended up being different enough that the number of overall shapes needed to increase (not the number of total shapes in the puzzle but the number of different sizes of shapes). This iteration also saw a slightly different approach to the

Figure 3-20 Martin House Iteration 2 - Abstracted Floorplan

Source: Author

level of abstraction. This iteration saw the removal of all compound shapes and a number of complex shapes where added (see the bottom-most terrace, kitchen, and the circulation that runs between the bedrooms). They were added in to give the player more of a direction when it came to arranging the shapes when solving the precedent puzzle. This addition of complex shapes helped to resolve the problem of overlapping shapes which arose in the first iteration for the Martin House.

Despite the changes made to the layout, no significant changes where made the actual process of how a player would go about solving a level. The only real change would be the removal of player-defined grouping from the *manipulation* stage of the user action cycle.

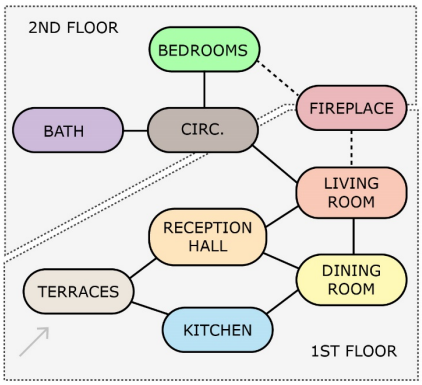


Figure 3-19 Martin House Iteration 2 – Function Bubble Diagram

Source: Author

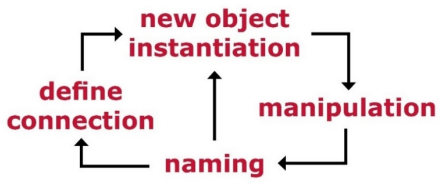


Figure 3-21 Martin House Iteration 2 - User Action Cycle

Source: Author

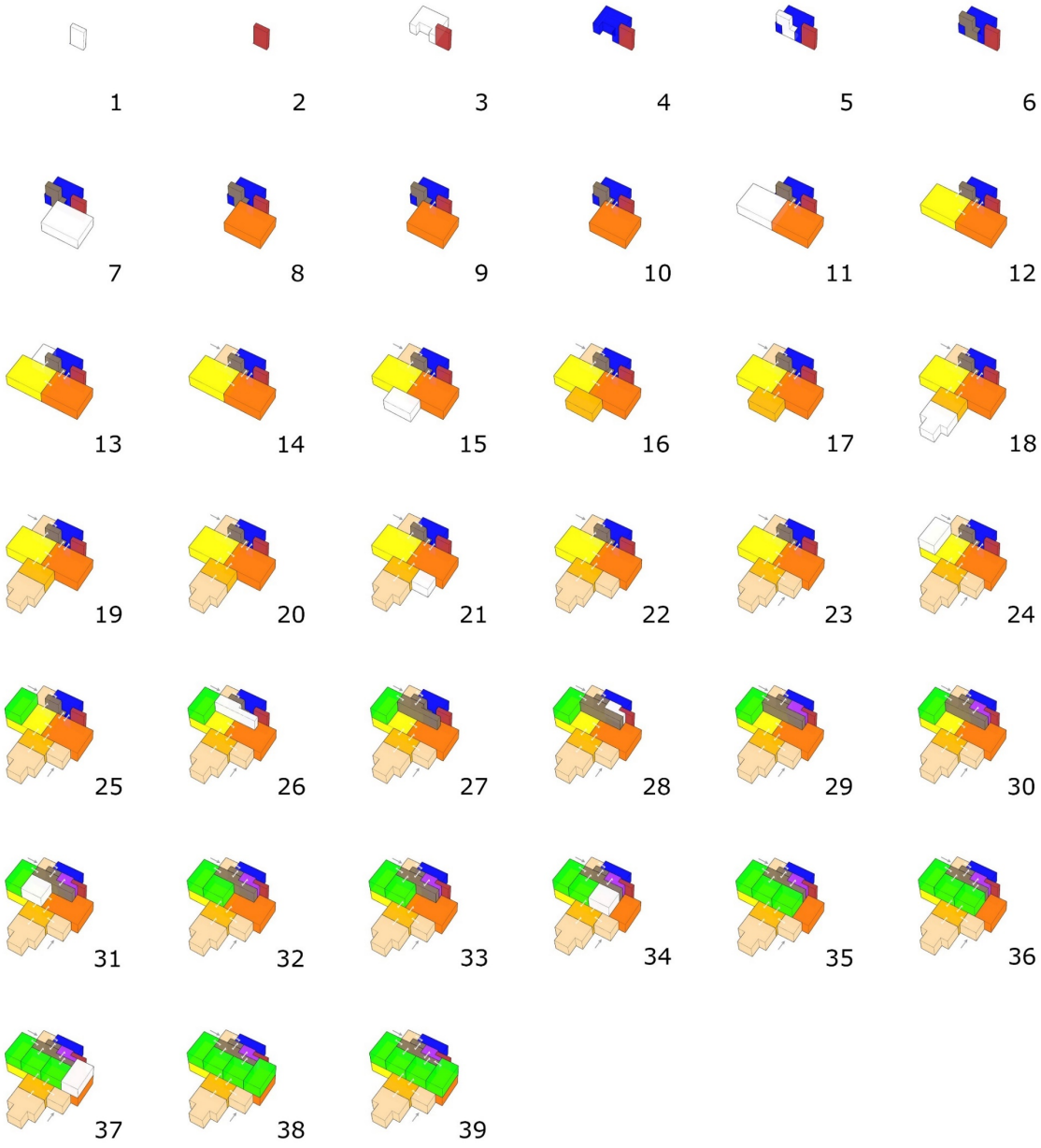


Figure 3-22 Martin House Iteration 2 - Original Solution

Source: Author

Some significant issues arose with this iteration. The most important being that in the attempt to simplify aspects of the abstracted layout, namely getting rid of compound shapes, made the process of reaching a solution a great deal harder. This difficulty is shown in the step-by-step instructions for one way of reaching the original solution for this iteration. If one were to place the fireplace first (which is typically seen as the correct move when dealing with any prairie house) in steps 1 and 2, then the only next step that could be made with any certainty on the player's

part would be to place the kitchen in steps 3 and 4, and not the living room as would be expected for a prairie style house.[17] This problem is due to the fact that the living room has no indication where it should connect to the fireplace, like if their corner points aligned or if the fireplace was perfectly centered along the length of the room. This problem involving the ambiguity of certain shapes' placements between eachother also has the unintended side effect of making the process of creating variations of the Martin House using this abstraction way harder than in the previous iteration. It was also found at this point that providing the player with unlabeled shapes also made it extremely difficult for them to start, as they had no real guiding direction or indication of what they were supposed to do.

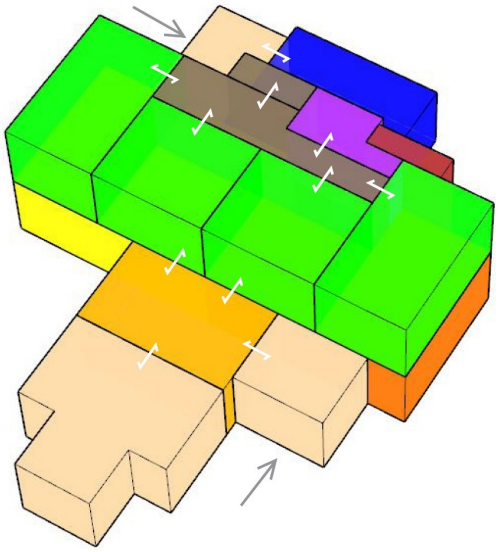


Figure 3-23 Martin House Iteration 2 - Original Arrangement
Source: Author

3.3.3 Iteration 3

This iteration worked to combine the lessons learned in both the first and second iterations. Compound shapes were added back in, and their use was expanded upon to help break up complex shapes (though some complex shapes still existed in this iteration) in addition to providing some distinguishing features like in the bedrooms where the division between the two shapes that are combined to make one of the bedrooms corresponds to location of the closets. When breaking down shapes to form compound shapes, commonalities between shapes were also taken into account, as while this iteration worked to resist the urge to render all shapes completely uniform, it still strived for enough uniformity throughout to assist in player generated variation. This iteration also saw several changes to correct some of the inaccuracies of the previous two abstractions. The reception hall was changed so that it would include the space between the living and dining room. A distinction was made between vertical and horizontal circulation as well. These alterations lead to a not insignificant alteration in the functional layout.

Figure 3-25 Martin House Iteration 3 - Abstracted Floorplan

Source: Author

A significant difference between the third iteration of the Martin House and all the ones that came before it was a shift in the process in which a player would go through the motions of solving a level. This change was brought about from the change in the information provided to the player. In the first two iterations, the player was provided with a set of shapes and a set of labels. They would then need to apply the labels the various shapes during gameplay as a part of the user action cycle. In this iteration, players are instead provided a set of labeled shapes, shapes that already have their room functions assigned to them thereby placing the naming action before the instantiation of a single object and therefore outside the user action cycle. The hope was that all of these alterations would help to provide the player with enough starting information to not feel completely overwhelmed at the beginning of the level.

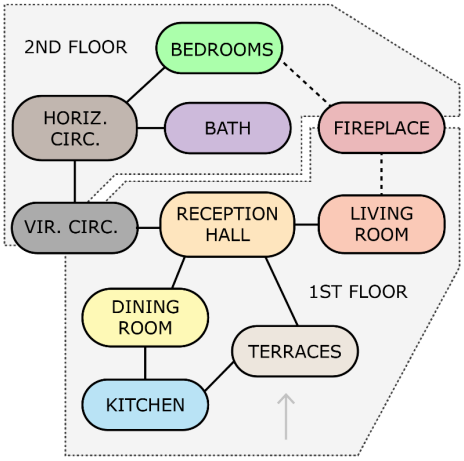


Figure 3-24 Martin House Iteration 3 - User Action Cycle

Source: Author

This iteration saw the most experimentation in terms of the generation of design variations when compared to the previous iterations and was found to be very promising. This experimentation and the specifics of its findings is detailed at the end

of this sub-section However, it is worth noting here that many of the adjustments made between iteration 2 and 3 were successful in achieving their desired affect when it came to the generation of variations and only a few minor adjustments were needed before it was ready for implementation within the *Unity* editor. It is worth noting that iteration 3 and not iteration 4 of the Martin House is used in parts of Chapter 4, specifically the parts dealing with design schema and the generation of design variations, and will be noted within the text and the captions whenever this occurs.

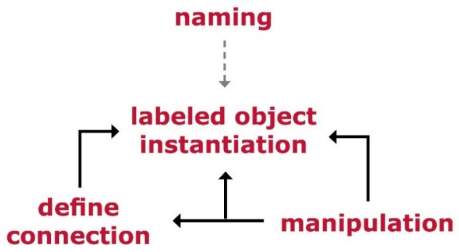


Figure 3-26 Martin House Iteration 3 – Function Bubble Diagram
Source: Author

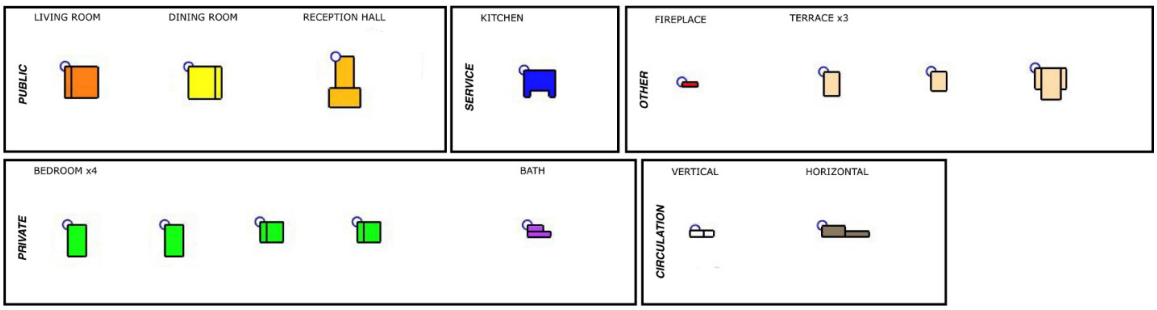


Figure 3-27 Martin House Iteration 3 - Shapes Provided to Player
Source: Author

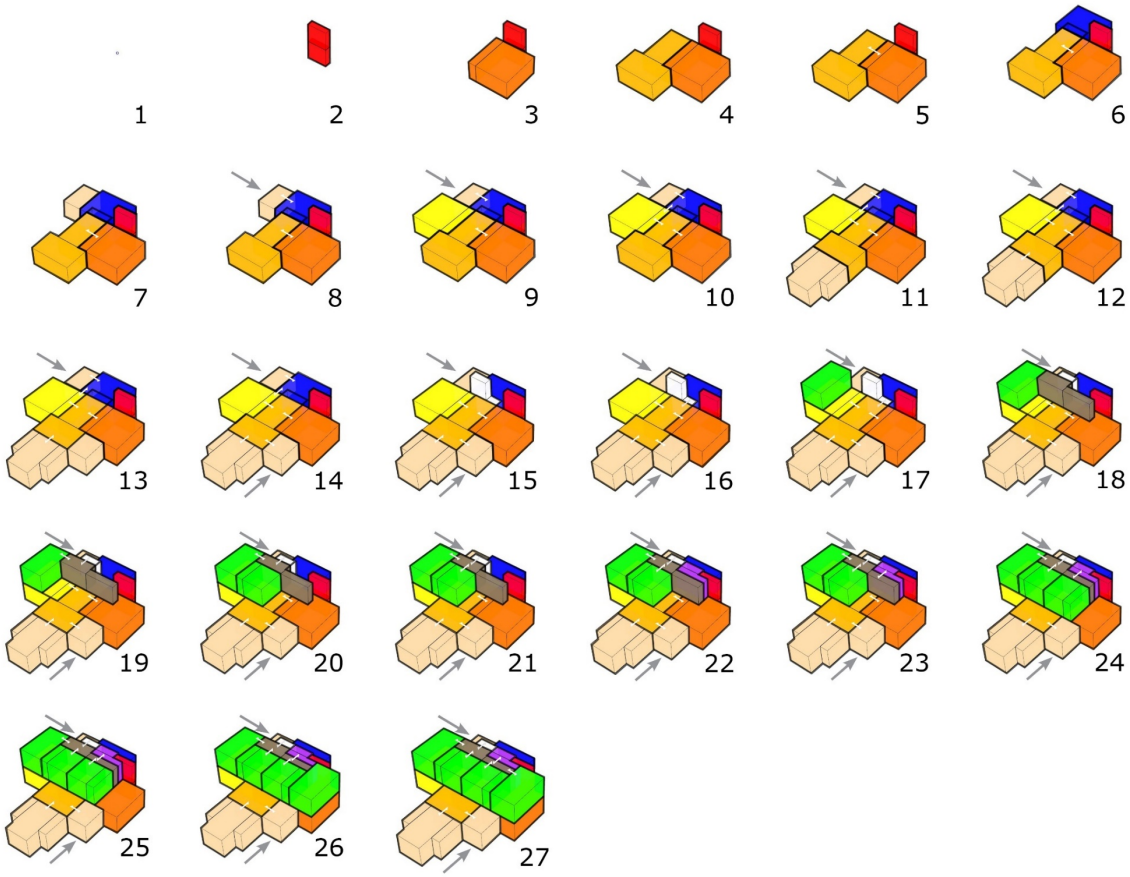


Figure 3-28 Martin House Iteration 3 - Original Solution

Source: Author

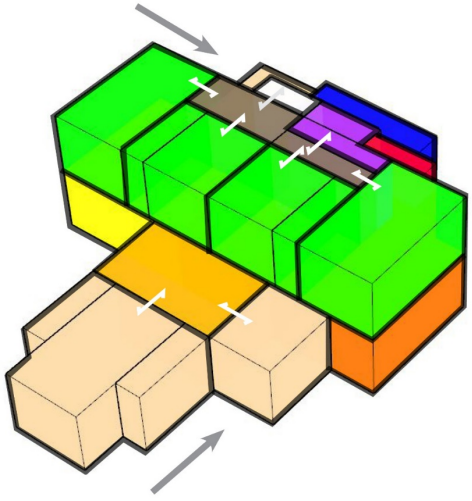


Figure 3-29 Martin House Iteration 3 – Original Arrangement

Source: Author

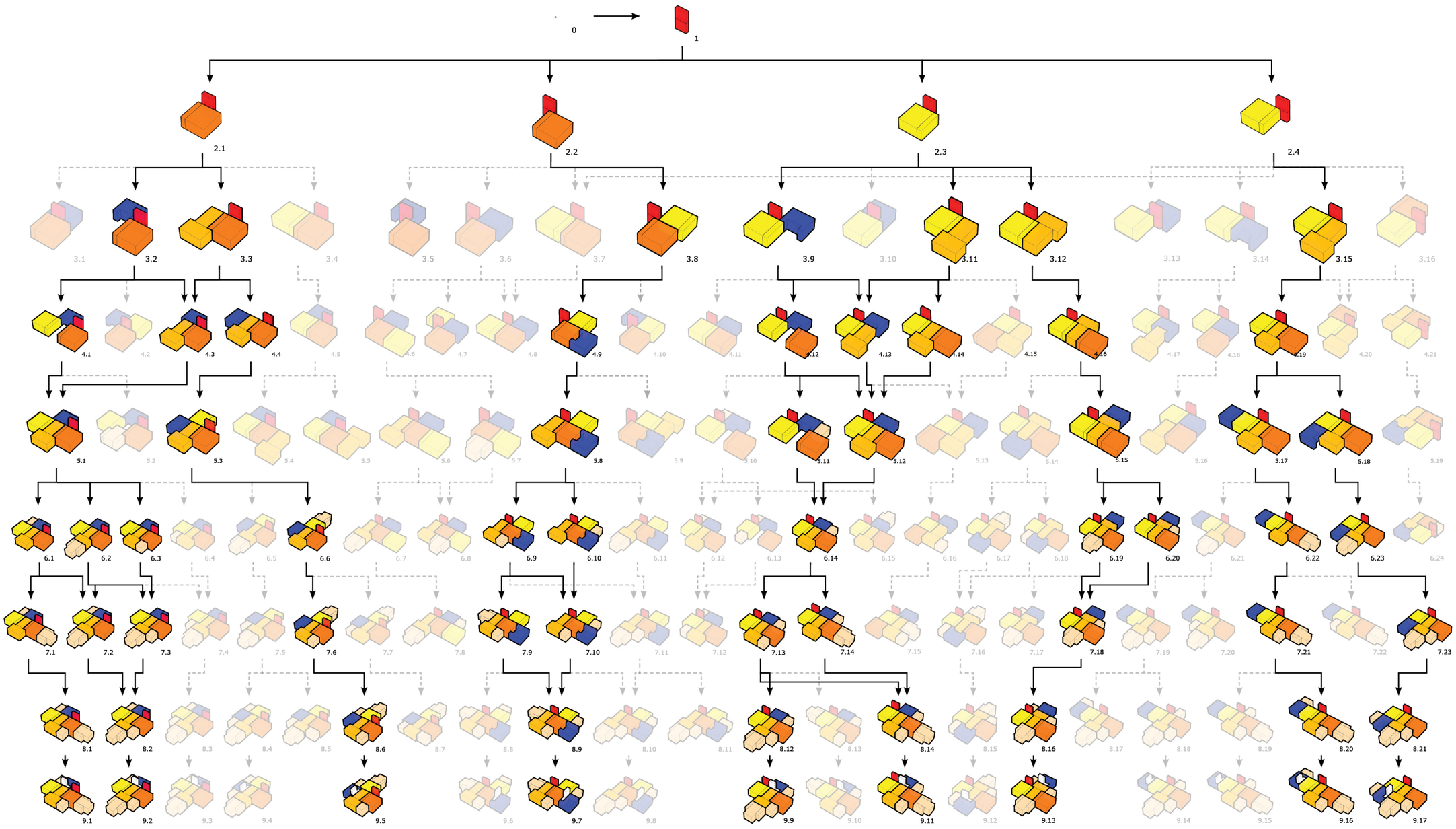


Figure 3-1 Testing Generation of Variations for Martin House Iteration 3 1st Floor

Source: Author

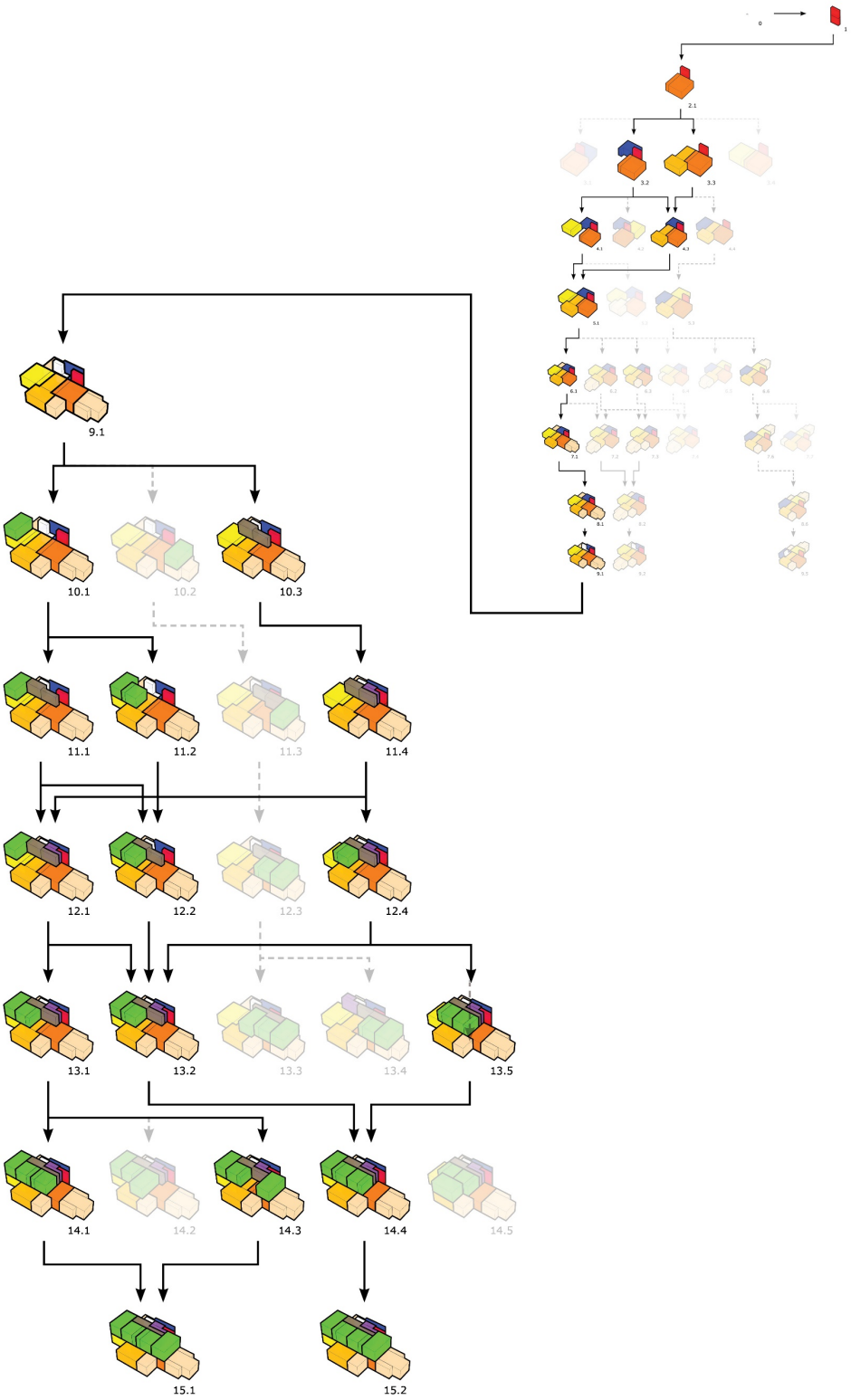


Figure 3-31 Testing Generation of Variations for Martin House Iteration 3 2nd Floor

Source: Author

3.3.4 Iteration 4

Figure 3-32 Martin House Iteration 4 – Abstracted Floorplan

Source: Author

Due to the progress made in iteration 3, it was decided to double down on the overall logic for abstracting and dividing up shapes used for most of the rooms in iteration 3 to two of the remaining rooms which still retained a tie over to the logic of iteration 2: the kitchen and its adjoining terrace. The kitchen, which in iterations 2 and 3 was a complex shape, was transitioned into a compound shape. The terrace, on the other hand, was made into a compound shape to help articulate the specifics of its form in the un-abstracted layout better, as well as to distinguishing some sub-shapes it had in common with other compound shapes.

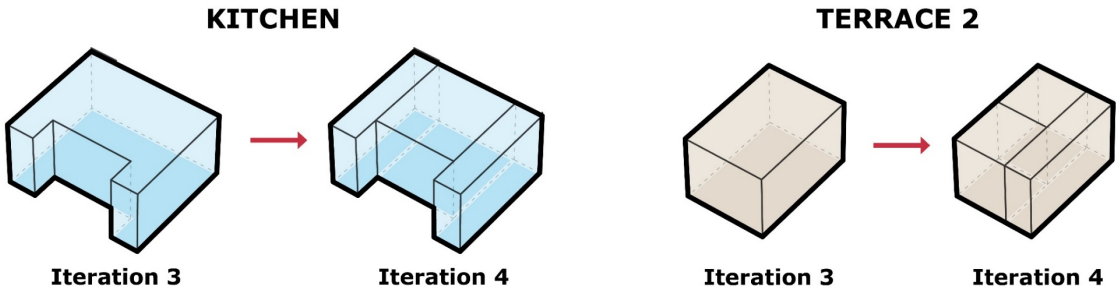


Figure 3-33 Martin House Iteration 4 Changes

Source: Author

This iteration of the Martin House maintains the same user action cycle that was used in iteration 3, where naming occures before object instantiation and is not

done by the player, as the player is provided labeled shapes instead of a set of shapes and another set of labels.

This iteration was implemented for this project and acts as the basis for the current level of this project's build of the proposed game. The specifics of how this abstracted iteration of the Martin House was translated into a video game puzzle is detailed in section 5.2 and in Appendix B. It is worth noting once again that iteration 3 and not iteration 4 of the Martin House is used in parts of Chapter 4 and will be noted within the text and the captions whenever this occurs.

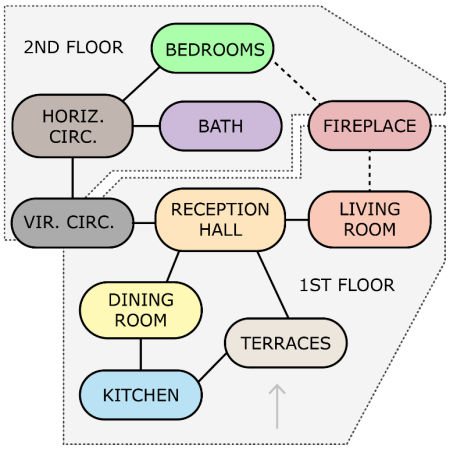


Figure 3-34 Martin House Iteration 4 Function Bubble Diagram
Source: Author

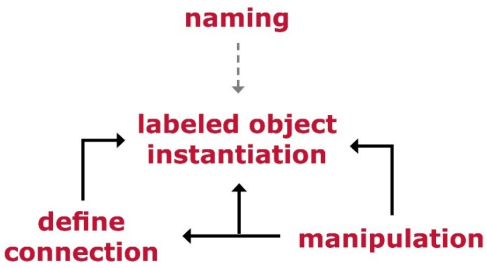


Figure 3-35 Martin House Iteration 4 – User Action Cycle
Source: Author

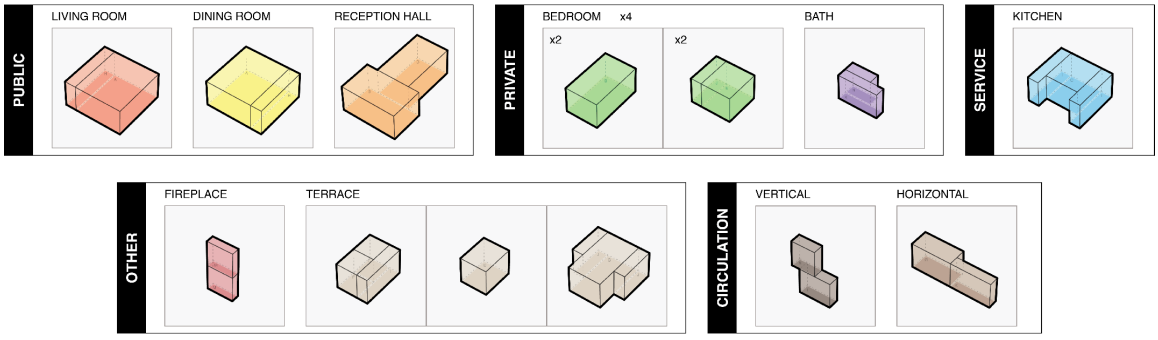


Figure 3-36 Martin House Iteration 4 – Labeled Shapes Provided to Player
Source: Author

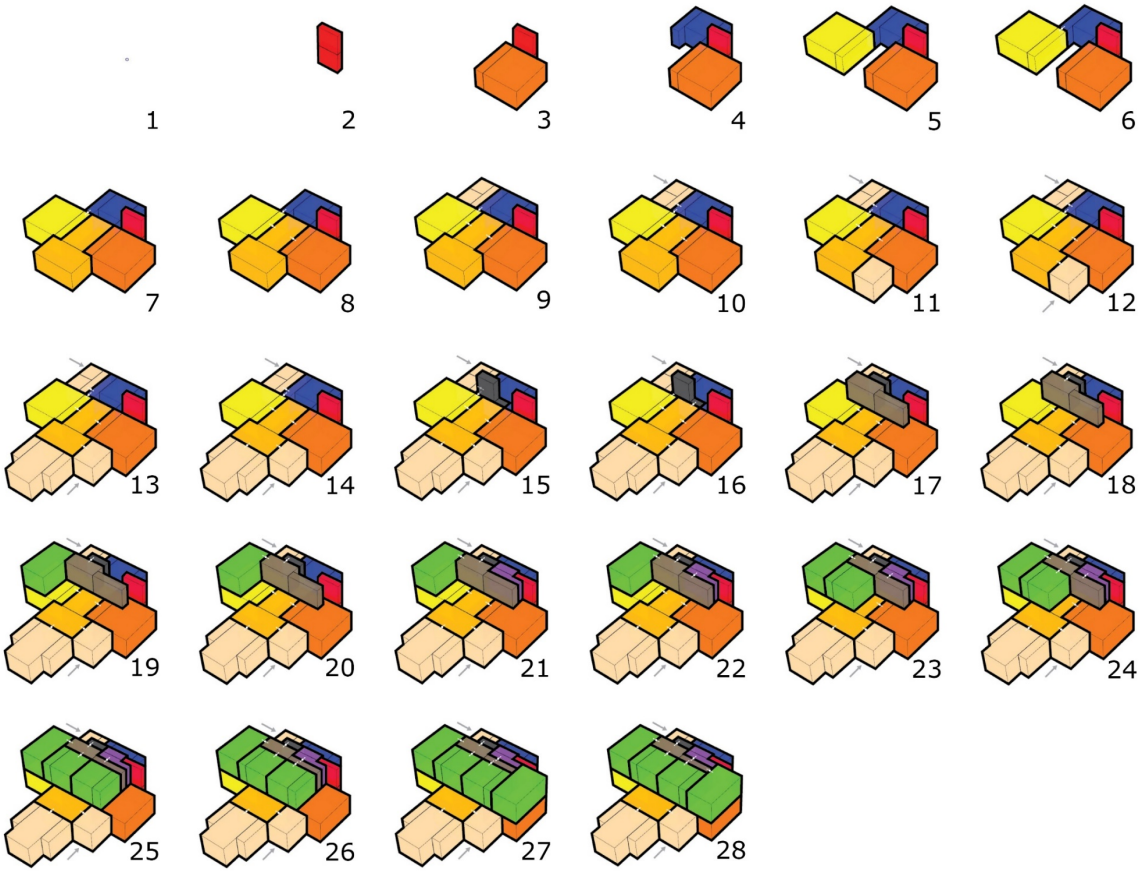


Figure 3-38 Martin House Iteration 4 – Original Solution

Source: Author

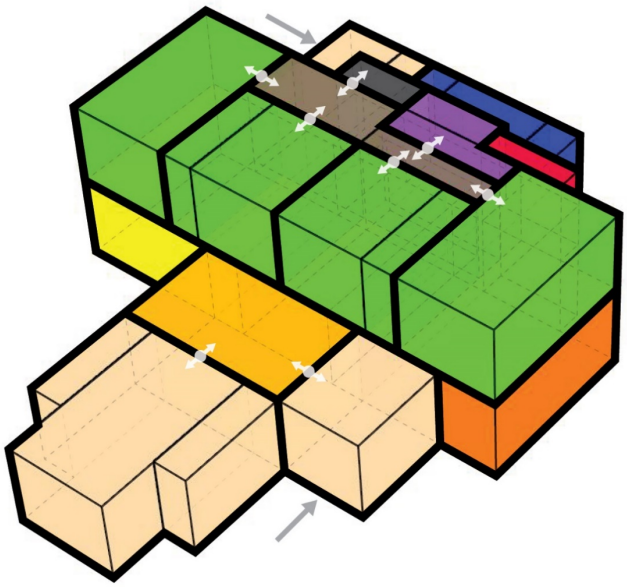


Figure 3-37 Martin House Iteration 4 – Arrangement

Source: Author

3.4 Abstraction: Villa Savoye

While the abstraction process for the Martin House was reasonably straight forward, the Villa Savoye proved to be a great deal more of a challenge than originally suspected when it was selected to be used for this project. Figure 3-39 shows an overview of the four different iterations, and as can be seen, while the first and third floors are reasonably straight forward, despite the presence of many unique shapes, the second floor's spatial division is relatively complex, despite the apparent simplicity expressed within the original floorplans.

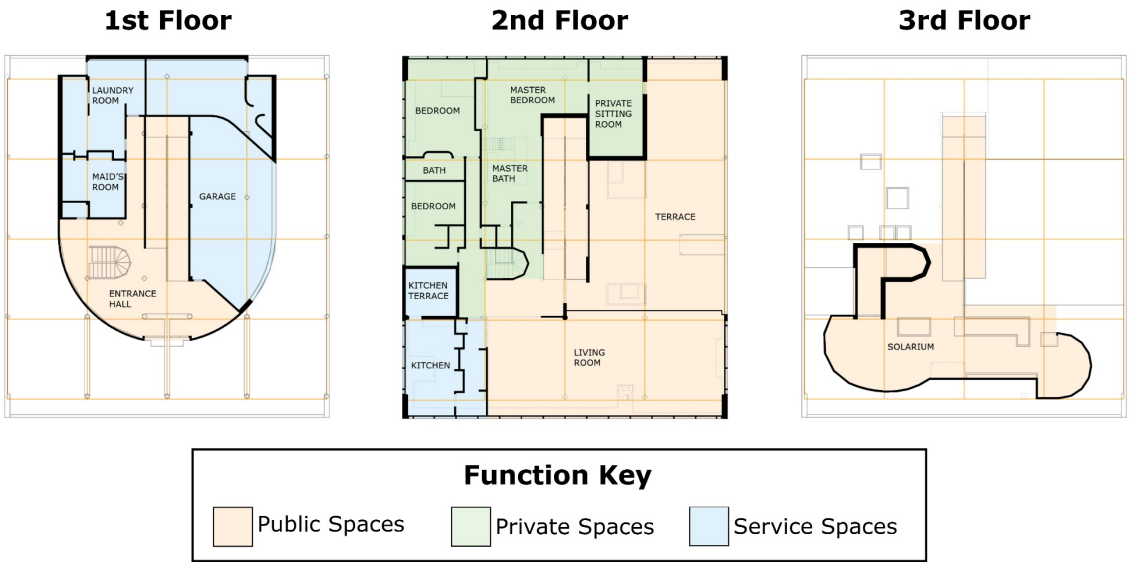


Figure 3-39 Villa Savoye Floor Plan

Source: Author

This complexity seems to be linked to the precedent layout's lack of commonality between shapes, with each shape seeming to only vary in size only slightly, enough to matter when it comes to whether or not the shapes will correctly line up, but not enough to be readily apparent to the human eye when simply looking at the shapes.

While a "final" iteration of the Villa Savoye was generated, not as much experimentation has been done on the ease in which one might arrive at a sensible design variation. The Villa Savoye has also not seen implementation within the current build at this point, though much of the preparation required outside the *Unity editor* has been done at this point.

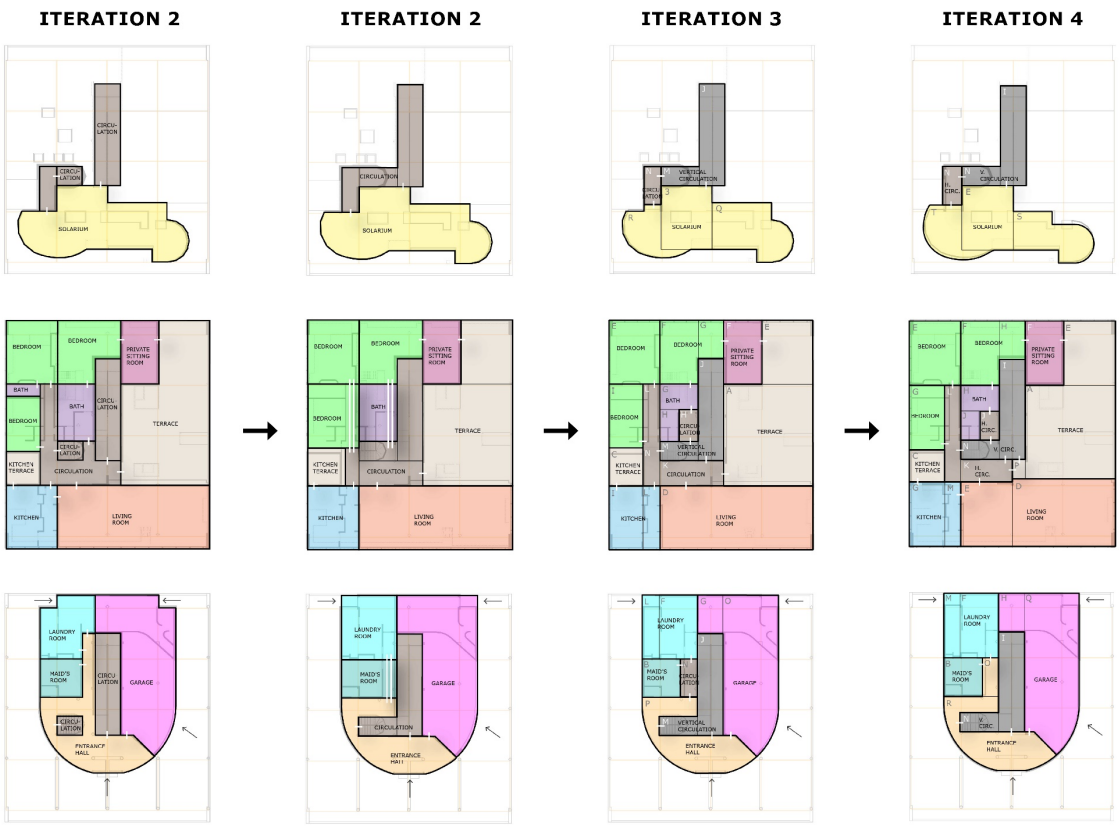


Figure 3-40 Villa Savoye Abstraction Iterations Overview

Source: Author

3.4.1 Iteration 1

This iteration of the Villa Savoye takes the place of the initial abstraction for the Villa Savoye, as while all the other precedents' abstractions started out as sketches on trace paper, the Villa Savoye abstraction has only ever existed within the digital. Because of this, a lot of the initial abstraction process was complicated due to the more precise nature of digital drawings. This essentially would mean that, unlike the Martin House which started very abstract and adopting more and more specific features with each iteration, the Villa Savoye abstractions will go back and forth from removing a lot of the identifying formal aspects and adding them back in.

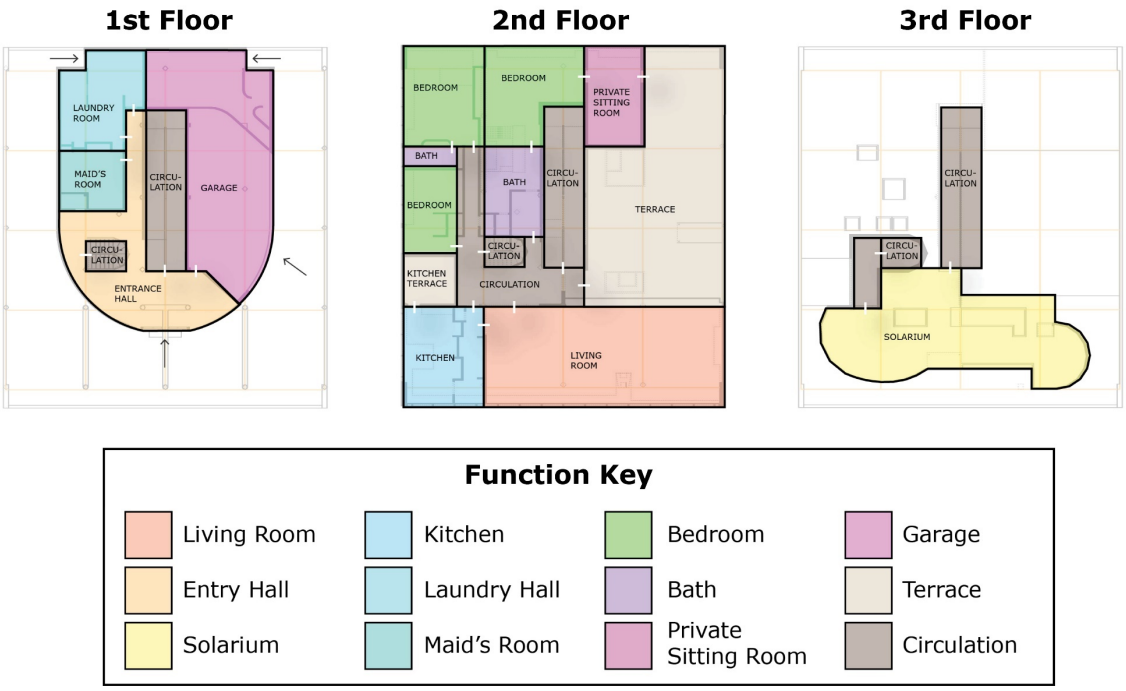


Figure 3-41 Villa Savoye Iteration 1 – Abstracted Floorplan

Source: Author

Similar to the first iteration for the Martin House, the first iteration for the Villa Savoye's abstraction has an almost identical user action cycle to scenario 3.1 and is identical to the user action cycle for the first iteration of the Martin House. Same as in the Martin House's iteration, in this iteration the player goes through a process *new object instantiation, manipulation, naming,* and *defining connections* until they have exhausted the list of provided shapes and room functions (subsection 3.3.1 details of what each of these steps within the action cycle).

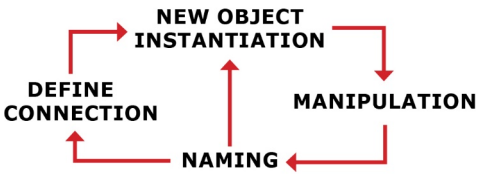


Figure 3-42 Villa Savoye Iteration 1 Action Cycle

Source: Author

Iteration one of the Villa Savoye ended up being a great deal more difficult to solve than was initially predicted, owing in large part to the sheer number of shapes which only varied slightly in their size; close enough to prevent shapes from aligning easily, but not different enough to be easily discernable by the human eye. This made the whole process of trying to assemble the Villa Savoye shapes a frustrating endeavor and this particular problem will continue to be an issue for this precedent for all the following iterations, with varying degrees of resolution.

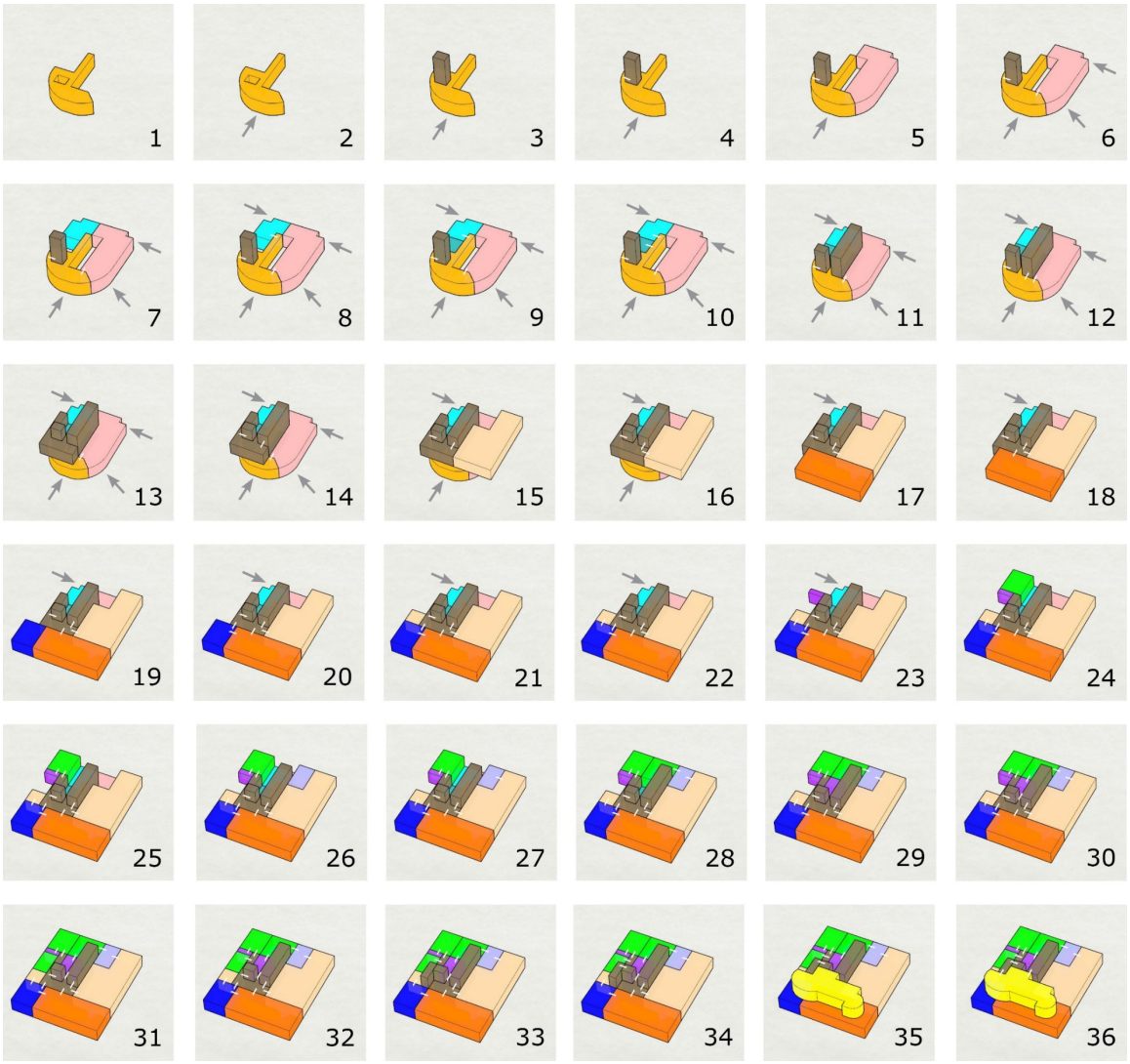


Figure 3-43 Villa Savoye Iteration 1 – Original Solution

Source: Author

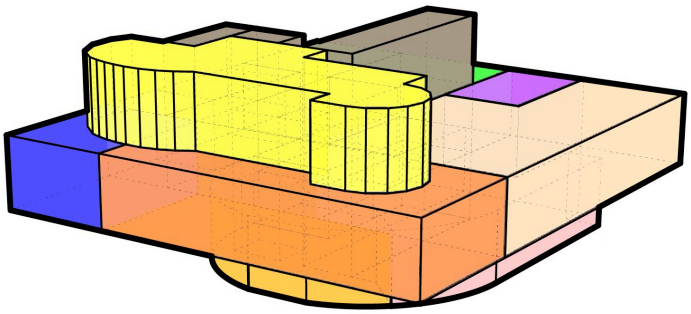


Figure 3-44 Villa Savoye Iteration 1 – Original Arrangement

Source: Author

3.4.2 Iteration 2

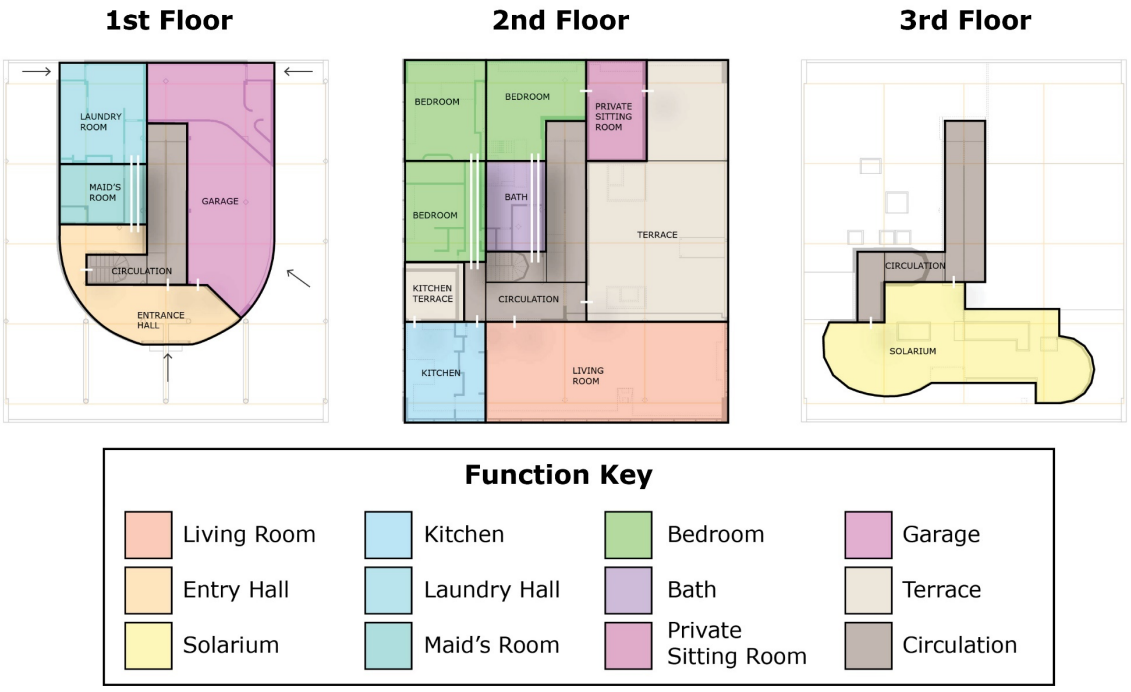


Figure 3-45 Villa Savoye Iteration 2 – Abstracted Floorplan

Source: Author

In iteration two, an effort was made to bring as much uniformity to the Villa Savoye's abstracted layout as possible, to counteract the frustration brought about by a lack of shard sizing in the previous iteration. A reduction in the number of rooms distinguished as separate entities, primarily by merging a circulation space with an adjacent room. The merging of rooms with adjacent circulation spaces would mean that a new connection type would need to be available to the player so they could, in some way, maintain the functionality of those removed circulation zones. This iteration also saw the addition of a small handful of compound shapes, again trying to bring as much sizing conformity into the abstraction of this design as possible.

Figure 3-46 Villa Savoye Iteration 2 –Action Cycle

Source: Author

Aside from the additional type of connection that would have to be

made available to the player to use, no other changes to the user action cycle are necessary between this iteration and the previous.

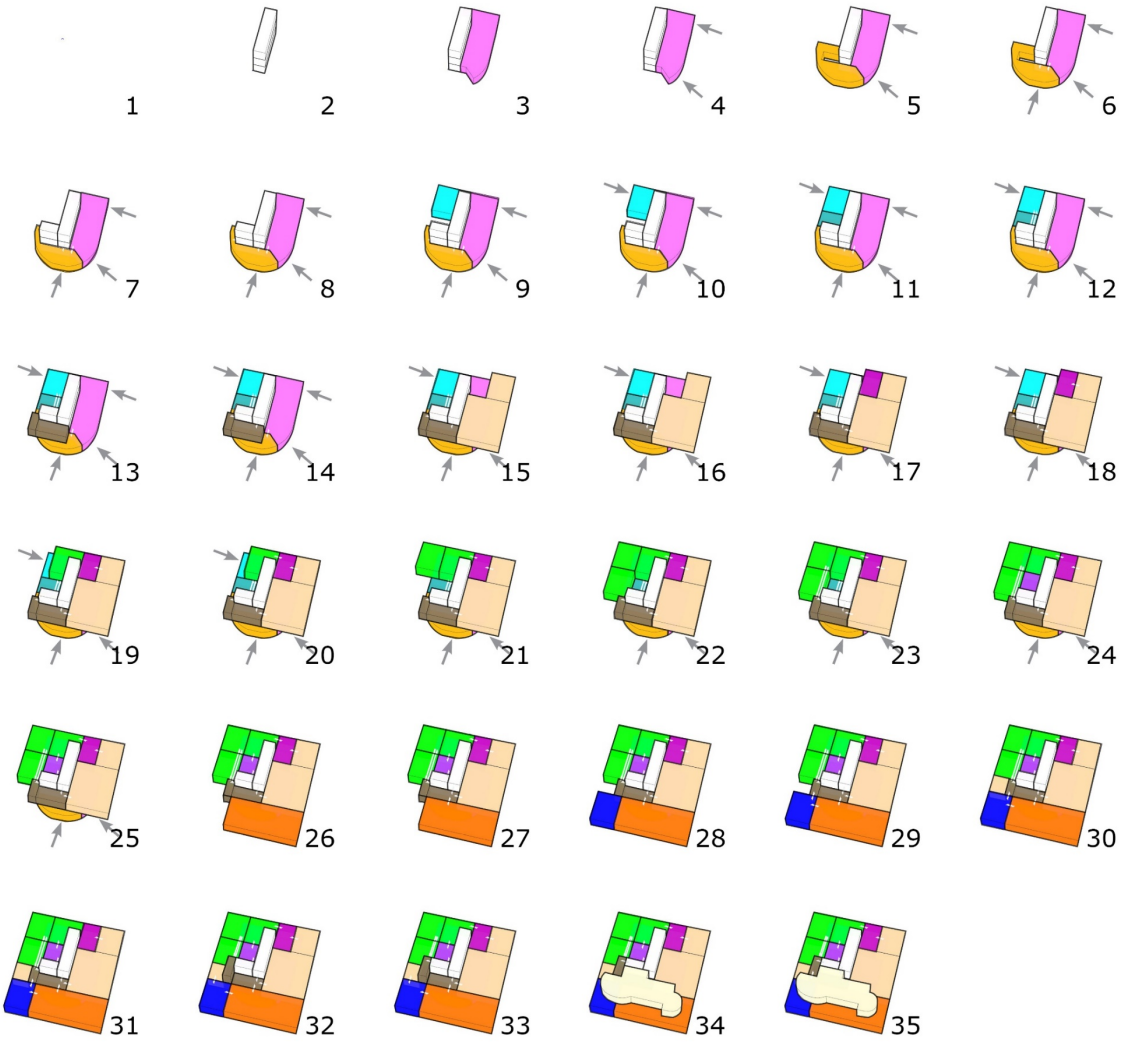


Figure 3-47 Villa Savoye Iteration 2 – Original Solution

Source: Author

While the addition of some sizing conformity did help reduce some of the frustration when arranging the shapes within this iteration of a potential Villa Savoye level, the loss of a number of the horizontal circulation spaces resulted in sWeveral rooms necessarily having connection points to rooms they did not feel like they should. There also was a weird disconnect between the specify of the unique shapes found in this precedent and the highly abstracted aspect of the second floor.

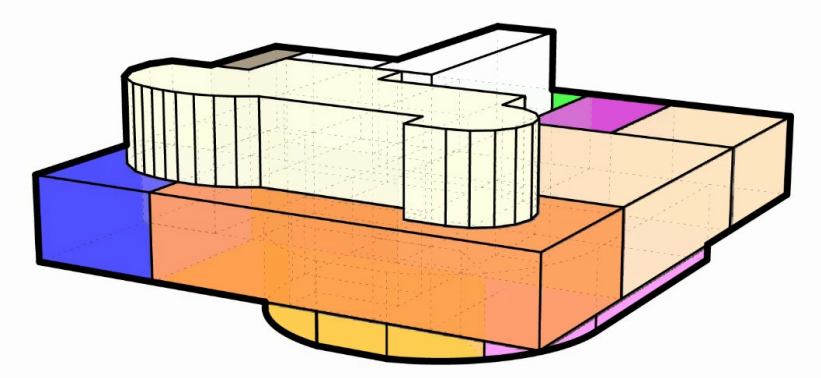


Figure 3-48 Villa Savoye Iteration 2 – Original Arrangement

Source: Author

3.4.3 Iteration 3

The third iteration sought a compromise between the first two iterations. All complex shapes were replaced with compound shapes to help with limiting the number of different shape sizes and bring more unity into the abstraction, while still maintain the specificity and necessary circulation functions that had been lost in the second iteration.

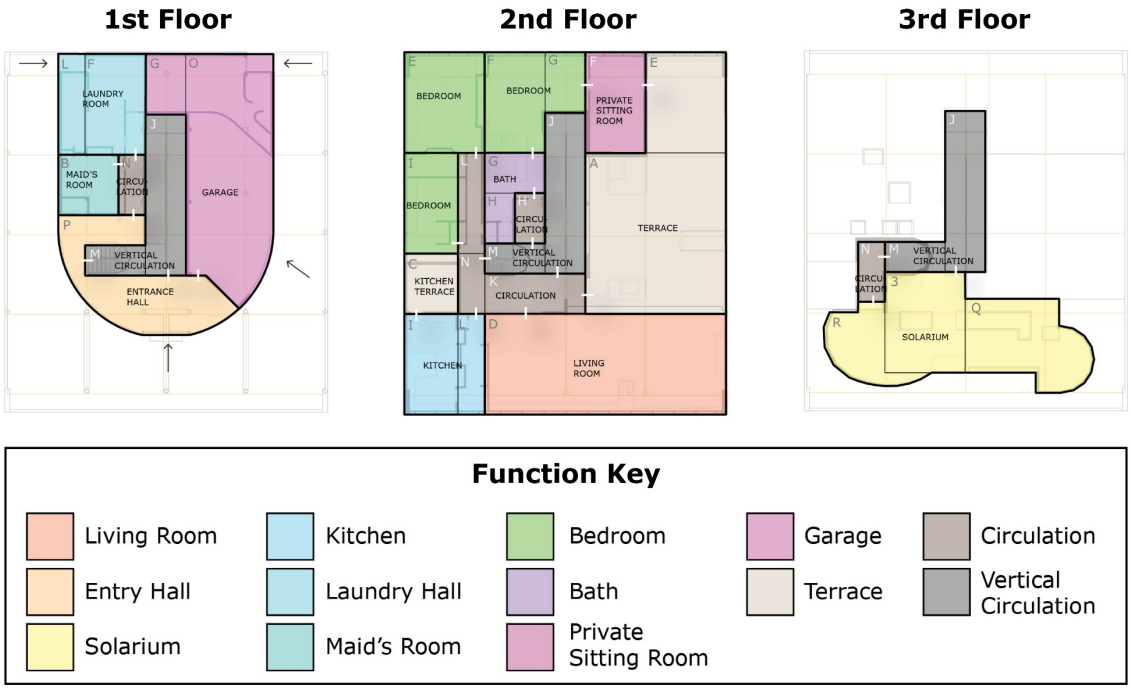


Figure 3-49 Villa Savoye Iteration 3 – Abstracted Floorplan

Source: Author

Much like in the third iteration of the Martin House, the third iteration of the Villa Savoye'ss abstraction saw a distinction made between the horizontal and vertical circulation spaces within the building.

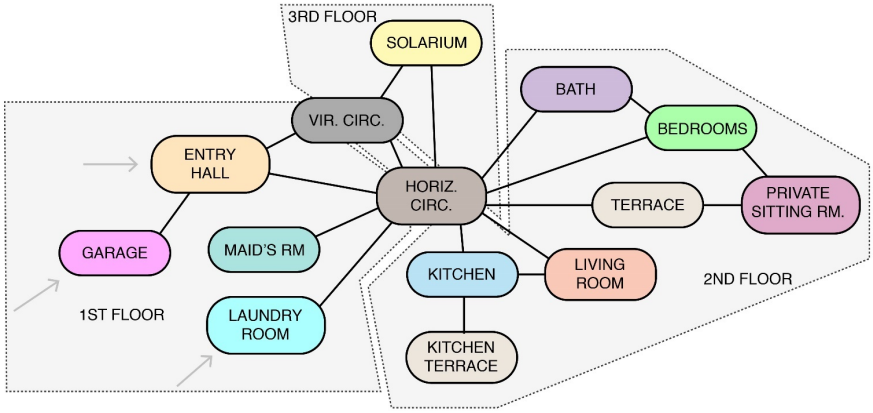


Figure 3-50 Villa Savoye Iteration 3 – Functional Bubble Diagram
Source: Author

Also similar to the third iteration of the Martin House, this iteration of the Villa Savoye saw the removal of the *naming* action from the user action cycle as the shapes provided to the player would come pre-labeled. With the sheer number of different rooms and room types found within this precedent, pre-labeling the shapes helped alleviate some of the confusion, and aided in distinguishing similarly sized shapes within the set of shapes provided.

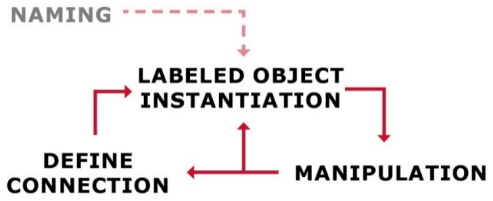


Figure 3-51 Villa Savoye Iteration 3 – User Action Cycle
Source: Author

This change to the user action cycle means that, while the player would not have quite as much freedom as they would have before, they are able to arrive at a final arrangement (be it the original arrangement or a variation) in fewer steps. The addition of more compound shapes, also had the effect of allowing the player a greater variety of paths they could take to arrive at the original arrangement, further reducing frustrating when moving the provided shapes around, and in placing them in relation to each other.

While this iteration saw great improvements from the previous two, it had a major problem: despite the fact that many of the shapes appeared to align with those around it, many of those shapes where anywhere from 2 to ¼ inches different. This difference, while small in size, was not so insignificant that it could be

overlooked when preparing the Villa Savoye for implementation. Therefore, at least one more of the Villa Savoye was necessary.

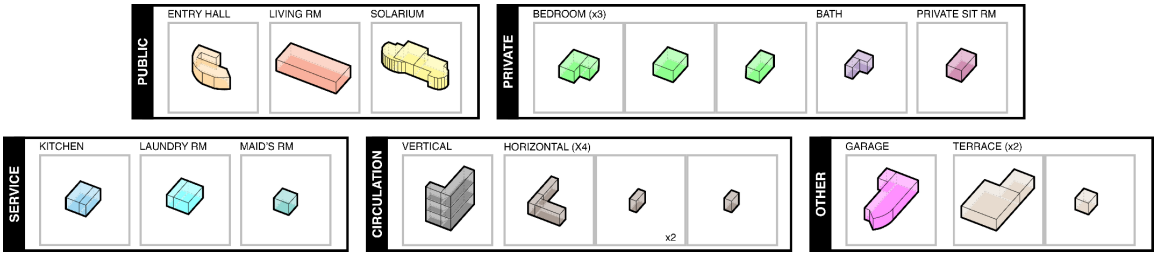


Figure 3-52 Villa Savoye Iteration 3 – Labeled Shapes Provided to Player

Source: Author

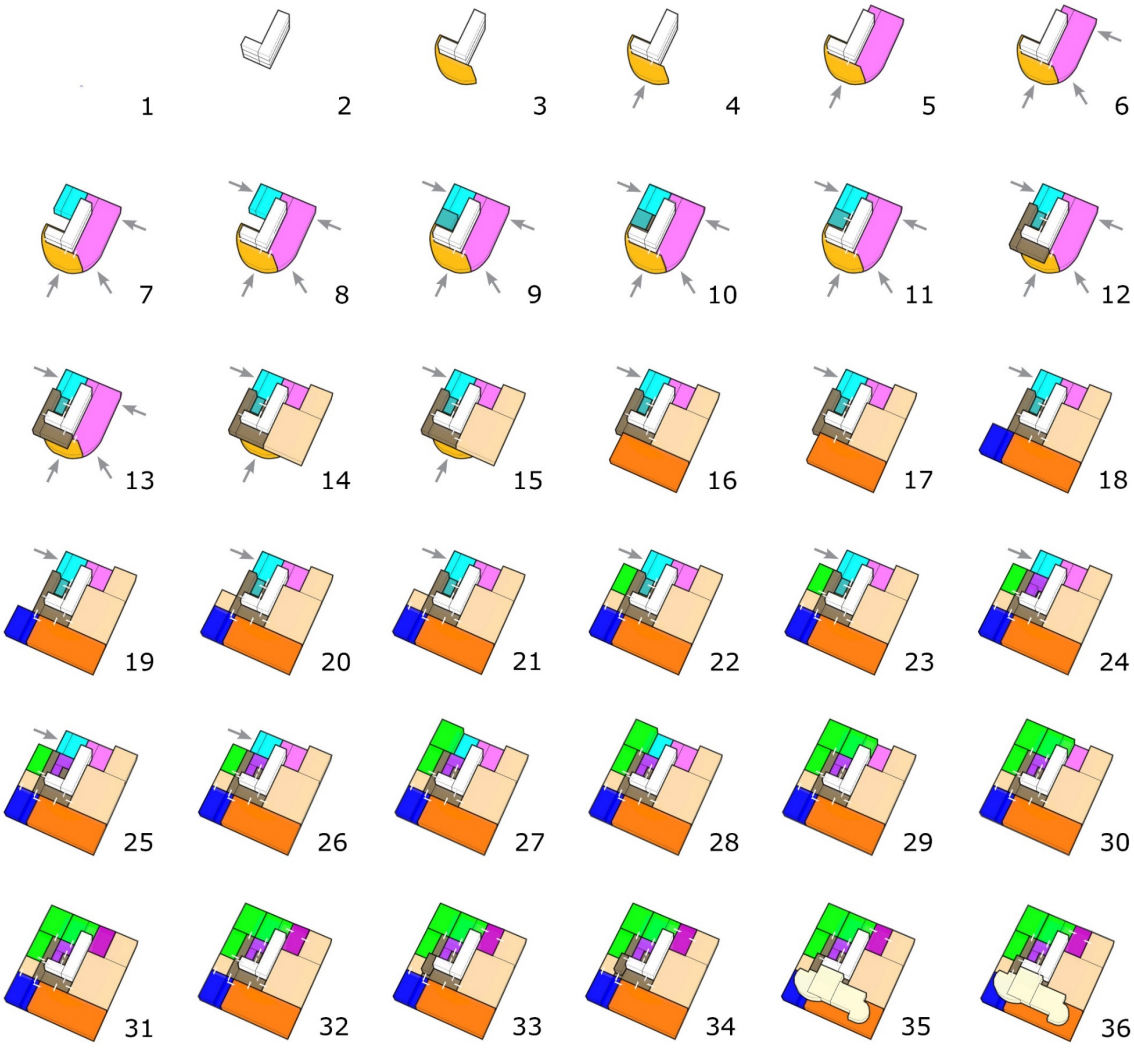


Figure 3-53 Villa Savoye Iteration 2 – Original Solution

Source: Author

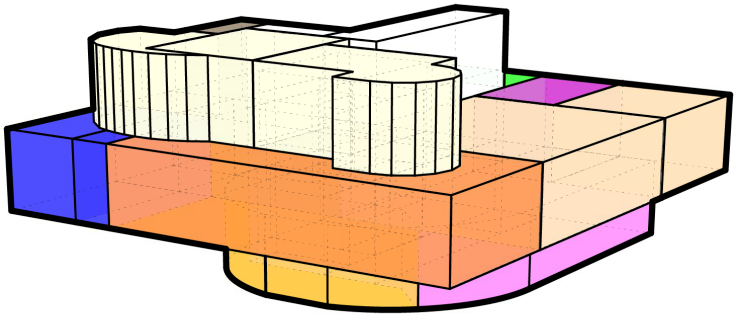


Figure 3-54 Villa Savoye Iteration 3 – Original Arrangement

Source: Author

3.4.4 Iteration 4

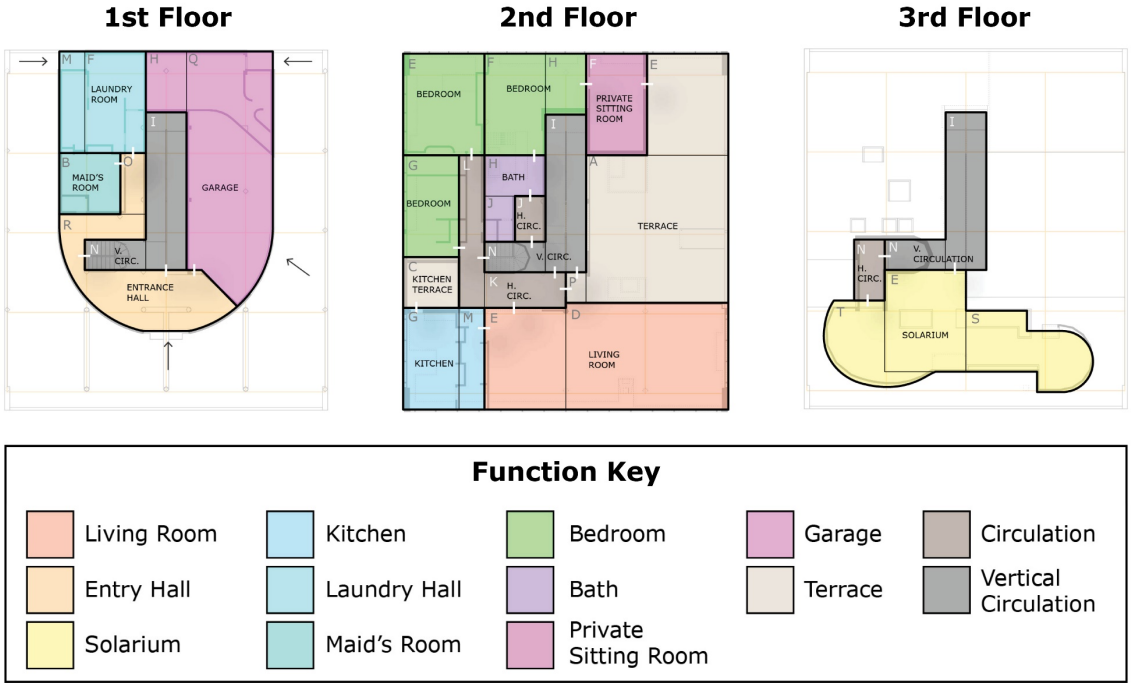


Figure 3-55 Villa Savoye Iteration 4 – Abstracted Floor Plan

Source: Author

As mentioned before, the previous iteration had problems with the shapes being slightly off. This iteration was created pretty much solely to address this issue. A few additional smaller changes to shapes and room definitions where made in the hopes of being more accurate to the precedent, while still working to reduce user

frustration. However, these changes where not as significant as the similar changes made between previous iterations.

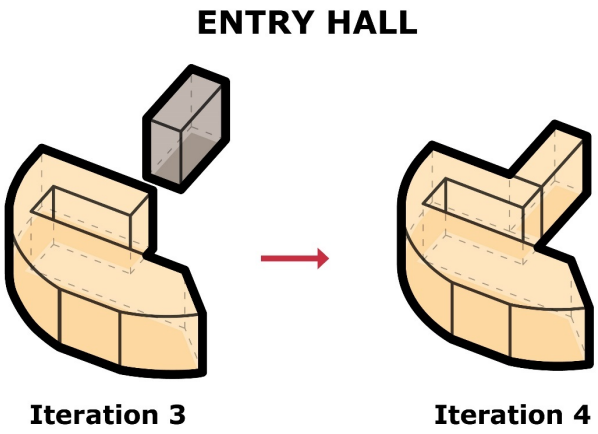


Figure 3-56 Entry Hall – Iteration 3 (left) vs. 4 (right)
Source: Author

The horizontal circulation on the first floor merged with the entrance hall, helping reduce the sheer number of small horizontal circulation rooms which were present in the third iteration. While there are still two tiny horizontal circulation rooms (located on the 2nd and 3rd floor), which are incredibly similar in size, those two had no clear alternative. The merging of the entrance hall and its adjacent horizontal circulation differs from the attempts to do something similar from iteration two as the circulation room/shape is being merged with a room type whose function is circulation adjacent (i.e. entrance hall) and not a completely separate function type altogether (i.e. maid's room).

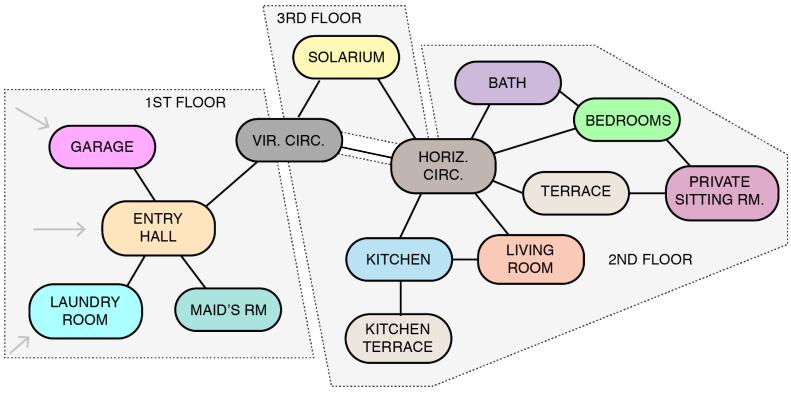


Figure 3-57 Villa Savoye Iteration 4 – Function Bubble Diagram
Source: Author

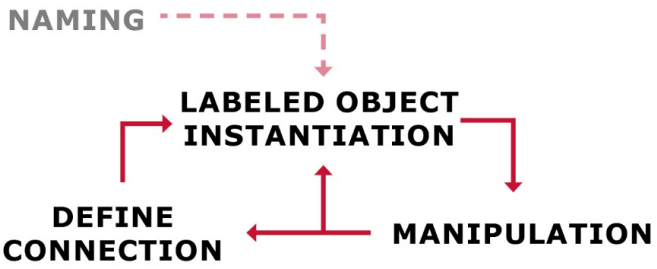


Figure 3-58 Villa Savoye Iteration 4 – User Action Cycle

Source: Author

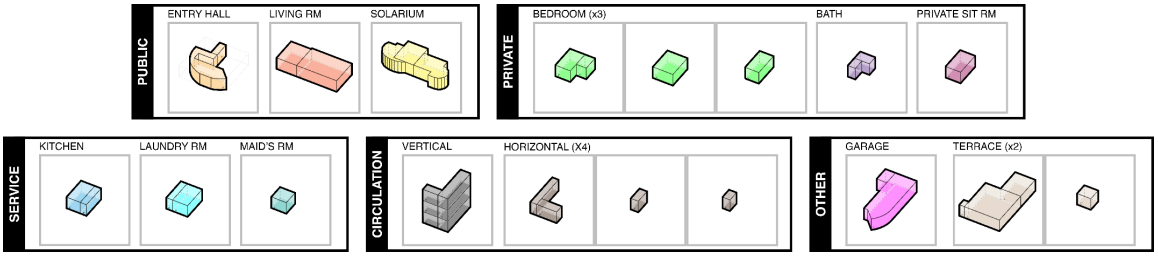


Figure 3-59 Villa Savoye Iteration 4 – Labeled Shapes Provided to Player

Source: Author

This iteration is the final iteration for the Villa Savoye abstraction, however, it is recommended that additional feasibility tests are done on this iteration to make sure it is not still overly complicated for anyone to put back together. That being said, it is uncertain how much room for improvement there actually is for a Villa Savoye abstraction and this iteration might be as good as it gets.

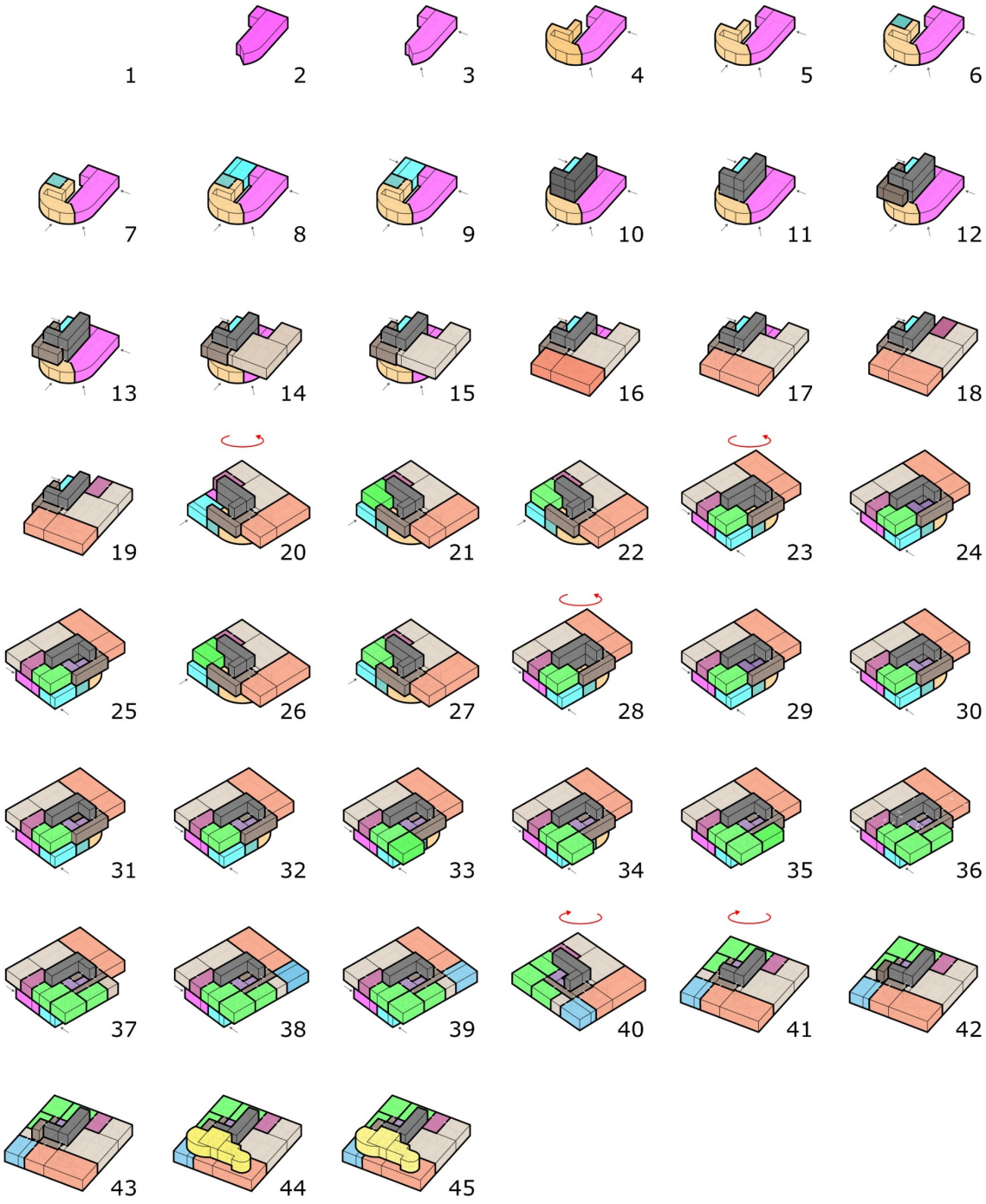


Figure 3-60 Villa Savoye Iteration 4 – Original Solution

Source: Author

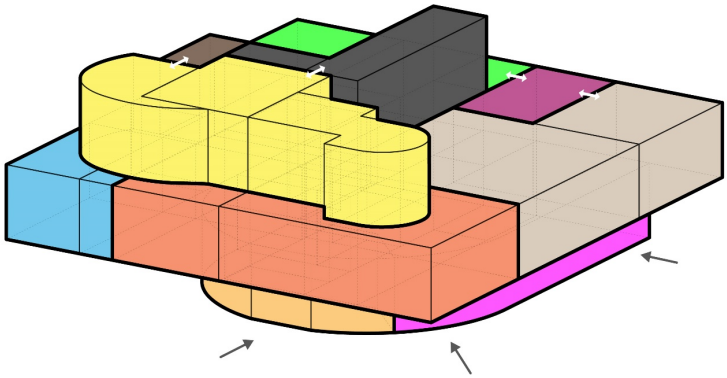


Figure 3-61 Villa Savoye Iteration 4 – Original Arrangement

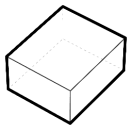
Source: Author

3.5 Abstracted Shapes

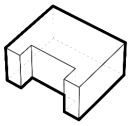
During the process of abstracting the layouts for the Martin House and the Villa Savoye, five overarching types of shapes where used: basic, complex, unique, compound, and unique compound. In this chapter I will go over each of these shape types, how these abstracted shapes relate to architecture and how architectural elements can be applied to them, the shapes used in the final abstractions for both the Martin House and Villa Savoye, and how different types of shapes can help or hinder a player's progress in solving each level.

3.5.1 Shape Types

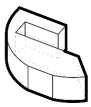
During the process of abstracting each precedent's floorplan detailed above, five main categories of shapes where used:



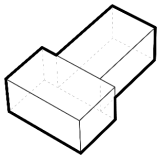
- Basic



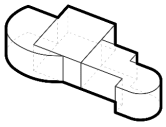
- Complex



- Unique



- Compound



- Unique Compound

However, the final iterations for both precedents only use four of these shape types: basic, unique, compound, and unique compound. In the following sub-sections, I will go over the basic reasoning behind each of the five shape types, explaining either why they are used or, in the case of complex shapes, why they were all replaced.

3.5.1.1 *Basic Shapes*

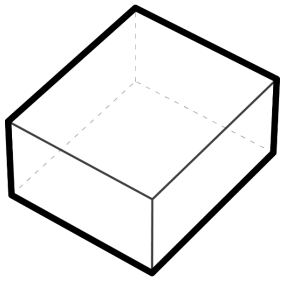


Figure 3-62 Basic Shape Example

These shapes consist of variously sized cuboids. They are the, as the name would suggest, the most basic and commonly used of the shapes, and the basic unit of division both precedent's final iterations' more complicated shapes wherever possible. They are prioritized as they assist in providing a higher degree of flexibility when rearranging the abstracted precedent shapes. However, their downside is that they are so abstract that they also make it hard to guide a player through the level, increasing the difficulty. In a sense, they can provide the player with more freedom in a way when compared to complex and unique shapes, but as that level of freedom goes up, difficulty also seems to increase. They also have the drawback of being too ambiguous in their appearance at times, again increasing the player difficulty when they try and solve the puzzle.

Due to the pros and cons detailed in the previous paragraph, basic shapes are only sometimes used on their own and are primarily used to form compound and unique compound shapes. The reasoning for this is detailed in the subsections dealing with those compound and unique compound shapes.

3.5.1.2 *Complex Shapes*

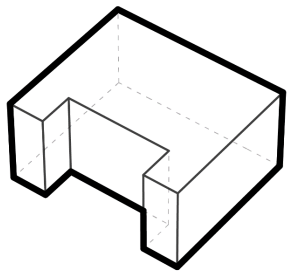


Figure 3-63 Complex Shape Example

Complex shapes fall somewhere between basic and unique shapes and can are used in some of the earlier iterations for both precedents. Complex shapes, like basic shapes, are comprised solely of flat rectilinear surfaces placed at right angles from each other but are not limited to being just a simple cuboid. Complex shapes were originally used to help bring some additional specificity and formal clarification to the abstraction. They were also used to help address some of the issues that had arisen in some of the preliminary abstractions where shapes ended up needing to be nested within one another, like in the Martin House case with the kitchen and circulation shapes. Compound shapes would eventually replace all of these complex shapes (see sub-section 3.5.1.4).

3.5.1.3 *Unique Shapes*

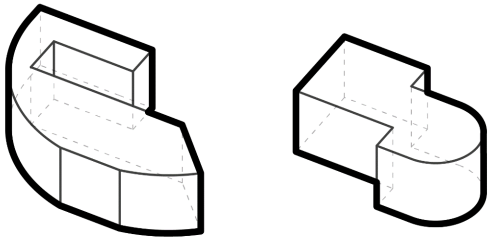


Figure 3-64 Unique Shape Examples Source: Author

When looking at precedents like the Villa Savoye and the Koshino House, it becomes apparent that the proposed game will need to address the curved rooms and their resulting shapes. Therefore, a strategy for addressing such rooms and formal aspects within a precedent's design during the abstraction phase is necessary. Unique shapes are more similar to complex shapes than basic ones; however, unlike complex shapes, unique shapes cannot be broken down solely into basic shapes and presents some trouble in the typical abstraction process. Currently, most iterations have dealt with unique shapes by performing a minimal amount of abstraction, allowing the shapes to maintain their own unique identity by

sticking more closely to their unabstracted counterpart through the use of curves while acknowledging that these curves themselves will limit the potential for player created design variations. Where possible, unique shapes are replaced with unique compound shapes described in Section 3.5.1.5 , with the goal of negating some of this limitation.

3.5.1.4 *Compound Shapes*

Compound shapes are comprised of two or more basic shapes. The basic shapes which comprise the compound shape remain visually distinct, but secondary to the overall compound shape. These basic shapes move together as one unit within the game scene and for all intents and purposes the compound shape as treated as if it were a unified whole.

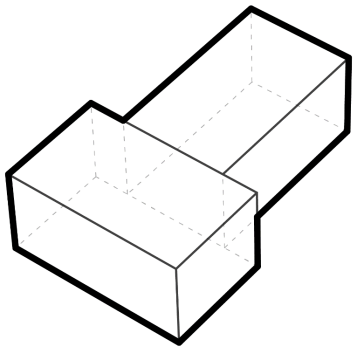


Figure 3-65 Compound Shape Example
Source: Author

Knowing when to use a compound shape is sometimes apparent, but frequently it requires careful consideration, both in selecting which shapes are to be compound as well as when determining what their basic shape components are in terms of sizing. The key reason and motivation behind using compound shapes should always be to one, increase usability on the player end, and two, increase accuracy to the precedent's unabstracted design.

There are three main things to look for when deciding when to use compound shapes and how to go about deriving the basic shape components developed from the initial testing done in Rhino during the abstraction process detailed earlier in this chapter The three things are as follow:

1. To break up complex shapes into its basic shape components.

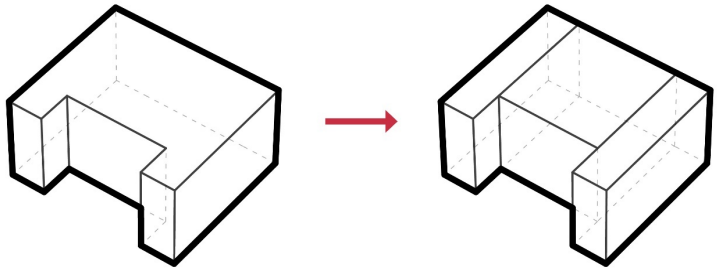


Figure 3-66 Breaking up Complex Shape into Compound Shape
Source: Author

2. To divide up a basic shape to account walls and other architectural elements that might affect the overall feel of a room or express an alignment between shapes and present within the overall composition of the precedent design.

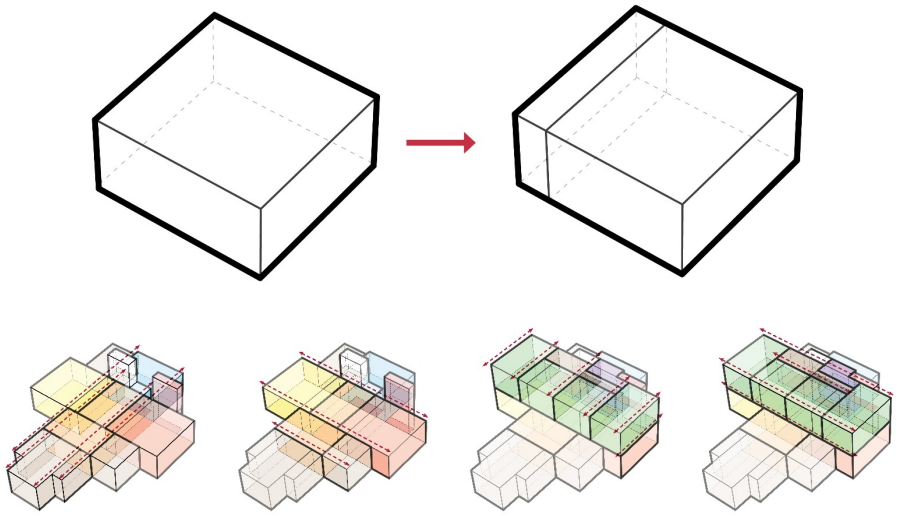


Figure 3-67 Using Compound Shapes to Aid in Shape Alignment
Source: Author

3. To pull out a basic shape located in multiple places throughout the design whenever such a division would make sense, typically by addressing one of the previous two.

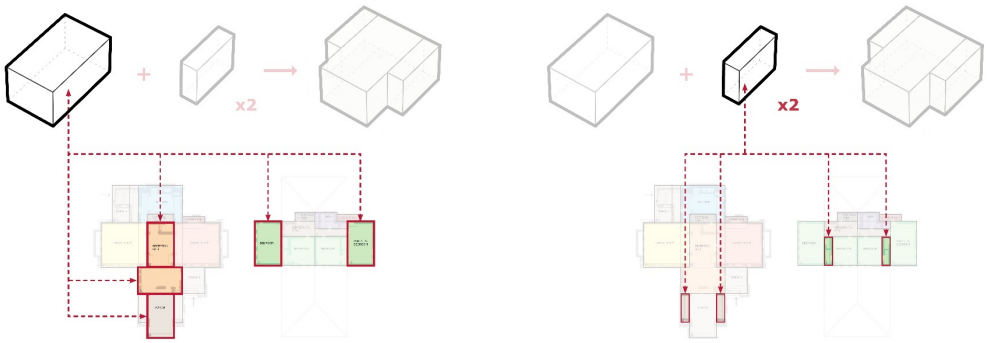


Figure 3-68 Creating Compound Shape via Pulling Out Common Basic Shape
Source: Author

3.5.1.5 *Unique Compound Shapes*

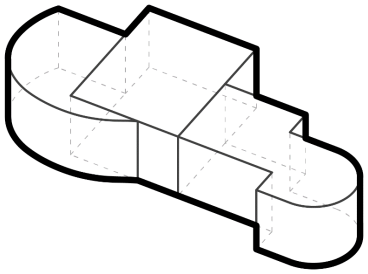


Figure 3-69 Unique Compound Shape Example
Source: Author

Unique compound shapes are precisely what they sound like: compound shapes in which are comprised of at least one unique shape. Unique compound shapes are typically, but not always a unique shape in which a common basic shape present elsewhere in the abstracted layout is subtracted out from the unique shape as a whole and then combined with the remaining shape/shapes to reform the original, but with the common basic shape made distinct from the unique shape as a whole. The primary reasons and things to look for when deciding where and how to use unique compound shapes when going through the abstraction process are otherwise identical to the ones used for regular compound shapes.

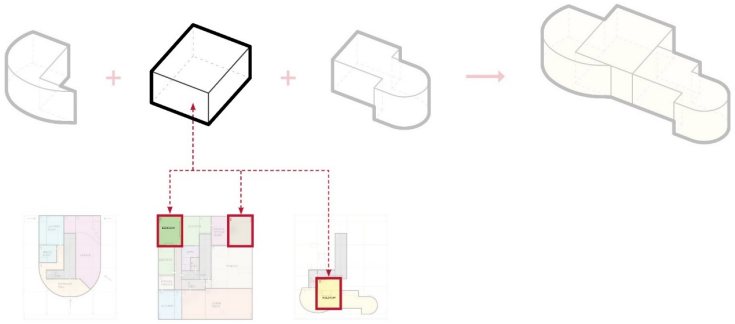


Figure 3-70 Creating Unique Compound Shape via Pulling Out Common Basic Shape
Source: Author

3.5.2 Shape as Placeholder

Up until now, shapes have been talked about as abstract representations of a precedents layout, a way of viewing the relationship between a precedent design and the abstracted design derived from it as it goes form architectural elements to abstract shape. How, then, can the relationship between the two, architectural and abstract, to be viewed the other way around: from abstract shape to architectural element? In the abstraction process, shapes are thought of as placeholders or massings for architectural elements. The shapes that the player arranges in the various level have the potential of being replaced with architectural elements after the player assembles them within the level. This replacement process could potentially take place either within the game program itself or in some other digital application (like Revit). In the latter case, the outcome of the level could be exported into another application (like Revit) where it would function as a massing model that could be used to place architectural elements.

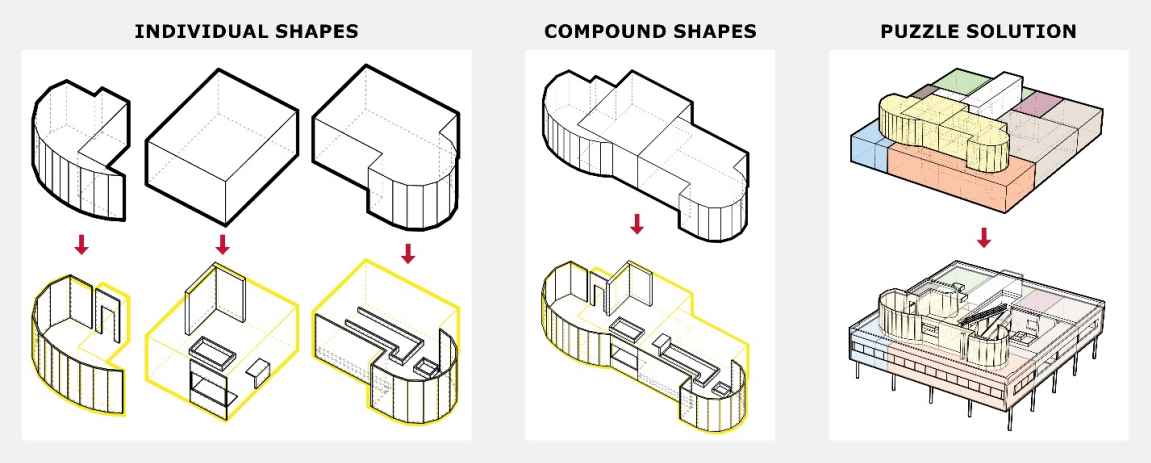


Figure 3-71 Shape as Placeholder to Architectural Elements

Source: Author

While this idea of moving backward through the abstraction process where abstracted shapes are replaced by architectural elements is not implemented within the game, the ideas it represents are essential to keep in mind when defining what these abstracted shapes represent. This idea of shape as placeholder works at the individual level where each basic or unique shape can be replaced with corresponding architectural elements. This replacement process can be repeated for each of the individual components of a compound shape. The goal of this being that once all shapes within a puzzle level solution are replaced with their corresponding

architectural elements, the player would have an architectural design. Once again, it is important to note that this idea is only theoretical at present.

3.6 Abstraction Methodology Summary

In order to create a 3D puzzle game, a method for deriving abstracted shapes from architectural precedents to function as the puzzle pieces for a level of the proposed game. This chapter details the development of two abstractions from two chosen precedents and the key takeaways from that process to be used as general guidelines in creating any future precedent puzzle levels. This project limited precedents to single-family housing. Project initially considered six different precedents for potential implementation as puzzle levels within an initial build of the proposed game, four belonging to the Prairie Style by Frank Lloyd Wright selected from Koning and Eizenberg's paper "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses" and two additional precedents from other styles that would contrast the Prairie Syle. The four Prairie Style homes were: the Martin House (Barton House), the Little House, the Roberts House, and the Willets House. Those four were then narrowed down to one, the Martin House. After narrowing down the Prairie Style houses to one, the selected Prairie house and two other precedents went through a preliminary abstraction process. These precedents were: the Martin House (Barton House) by Frank Lloyd Wright, the Koshino House by Tadao Ando, and the Villa Savoye by Le Corbusier.

After going through a preliminary examination, the list of three precedents was narrowed down to two: the Martin House and the Villa Savoye. Both precedents when through four different iterations of abstractions. After the creation of each of the abstractions for all iterations for both precedents, initial testing was done using either physical models or digital models using Rhino to test how easy the abstraction was to put back together if disassembled. When significant problems with the abstraction were found during this testing, another abstraction was created to try and address those concerns. Both precedents saw a total of four iterations.

Along with producing a blueprint for the puzzle pieces required in the implementation of the puzzle levels for both precedents, the iterative abstraction process served as a means of fleshing out how abstraction should take place and the types of shapes to be used. When abstracting the Martin House and the Villa Savoye, five total shape types are used: *basic, complex, unique, compound,* and *unique*

compound, with the fourth iteration for both the Martin House and the Villa Savoye dropping the *complex* shape type. While there are no hard, step-by-step rules for how to go about abstracting architectural precedents for implementation into this proposed game, this chapter, in exploring the specifics of how both the Martin House and the Villa Savoye were abstracted for this project, outlines some general guidelines to be kept in mind for any future work done for this game in terms of abstraction.

---

[1] Stiny, "Weights."

[2] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[3] Tadao Andō, "Tadao Andō : details," in *Andō Tadao diterushū* (Tokyo: Tokyo : A.D.A. Edita, 1991); Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Jacques Sbriglio, *Le Corbusier : the Villa Savoye*, Villa Savoye, (Paris

Basel, Switzerland : Fondation Le Corbusier : Birkhäuser, 2008); Frank Lloyd Wright, *Frank Lloyd Wright: the early work* (New York: New York, Horizon Press, 1968).

[4] Dixie Legler Guerrero, "Prairie style : houses and gardens by Frank Lloyd Wright and the Prairie School," ed. Christian Korab and Frank Lloyd Wright (New York: New York : Stewart, Tabori & Chang, 1999); Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Lee, Ostwald, and Gu, "A Combined Plan Graph and Massing Grammar Approach to Frank Lloyd Wright's Prairie Architecture."; "Spotlight: Frank Lloyd Wright," ArchDaily, 2018, accessed March 10, 2019, https://www.archdaily.com/513642/happy-birthday-frank-lloyd-wright.

[5] Guerrero, "Prairie style : houses and gardens by Frank Lloyd Wright and the Prairie School."

[6] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[7] Wright, *Frank Lloyd Wright: the early work*.

[8] "Martin House Complex." accessed December 02, 2018. https://franklloydwright.org/site/martin-house-complex/.

[9] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Wright, *Frank Lloyd Wright: the early work*.

[10] Andō, "Tadao Andō : details."

[11] Andō, "Tadao Andō : details."

[12] Andō, "Tadao Andō : details."

[13] ibidSbriglio, *Le Corbusier : the Villa Savoye*.

[14] Sbriglio, *Le Corbusier : the Villa Savoye*.

[15] ibidSbriglio, *Le Corbusier : the Villa Savoye*.

[16] Guerrero, "Prairie style : houses and gardens by Frank Lloyd Wright and the Prairie School."; Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

[17] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

Chapter 4: Gameplay

This dissertation proposes a game which would serve two purposes: one at the player level and one at the researcher/developer level, both looking at different key motivational factors. At the player level, the game would serve as an educational tool for the player to study architectural precedents, their styles, vocabularies, and the design rules used by architects, providing the player with a kind of tacit knowledge of design. At the researcher/developer level, the game would serve as a tool for larger scale data collection on the player's step-by-step process as they progress through the levels.[1] This chapter will primarily focus on how a player would experience and play the proposed precedent puzzle game and the features that would be needed to support such play.

The first two sections will briefly go ideas and concepts that have been looked and are important to the wider context of what this game strives to be but fall outside of the scope this project is looking at in terms of implementation within this prototype build. Section 4.1 looks at how variations in player motivations would affect how players approach video games. Section 4.2 is about the fact that this game is intended to contain a variety of different precedent puzzles contained within a library of precedents for the player to solve.

After those two sections, most of the remaining chapter will outline different gameplay features in the current build. Section 4.3 outlines the player action cycle and sections 4.4 through 4.6 outline the core actions in that cycle, *instantiation, manipulation,* and *connection*. Section 4.7 looks at how players are provided meaningful feedback on their choices through a scoring system. How the game tracks player choices and how that information can be accessed and used by the player is talked about in section 4.8. Section 4.9 discusses how the game accounts for player mistakes and provides a way for the player to backtrack, undoing previous choices with section 4.10 detailing how that can be done on a bigger scale within the *exploration* action cycle state. The last section of this chapter, section 4.11 discusses aspects of gameplay that have been seen to effect how easy or hard a puzzle is outside of the individual shapes themselves.

Many of the gameplay features talked about in this chapter have already been implemented within the current build of the game and their implementation is detailed in Chapter 5: Implementation.

4.1 Player Motivations

When looking at game design and software development in general, it is important to consider the motivations of the user, or in this case, the player. By noting the player's motivations are developers are able to ground the project in something more concrete.[2] This was looked at briefly in Chapter 2 where each scenario defined a user, their motivation, and the application's purpose.

A player's motivation when playing a video game can vary depending on the type of player and what kind of experience they are looking for in their gaming experience. This variation can alter what players are looking for in a game.[3] In terms of this project, by looking at different player motivations we can start to think of how a player would approach 3D spatial puzzles in a video game setting. For example, one player might prioritize obtaining the highest score possible within the level and might want to achieve this by solving the puzzle as intended, or by any means necessary (by exploiting bugs). Another player might not even care about getting a good score, and instead merely wants to be free in the forms they can make, using the puzzles as a form of self-expression and design exploration. Another player still might care more about sharing their experience with other players, caring more about the social element. These various player motivations can be thought of as frames which in turn can be used when brainstorming and developing potential features within the game.

4.2 Library of Precedent Puzzles

The video game proposed by this dissertation is a 3D educational puzzle game where the puzzles are based off a variety of architectural precedents. All the puzzles, when looked at as a group, comprise the *library of precedent puzzles*. In this way each game level/scene is like a book whose content is the precedent puzzle and the video game itself and how the different scenes are accessed by the player is like a library.

Due to the limited scope of this particular project, only one precedent puzzle implemented, the current library only has one book to continue the metaphor. However, plans have been drawn up for a second puzzle level/scene (the Villa Savoye) and there are plans to generate more puzzles for the library, all of which is discussed in greater lengths in Chapter 6: Discussion & Framework for Future Development, section 6.3.

4.3 Player Action Cycle

At the start of each game level the player is provided a set of 3D puzzle pieces and in order to complete the level the player uses a series of actions to arrive at a solution through placing and arranging these puzzle pieces within a 3D digital environment. Those actions form the action cycle first outlined in the project scenario, detailed in chapter 3, and further developed during the abstraction process for the Martin House and Villa Savoye, detailed in chapter 4.

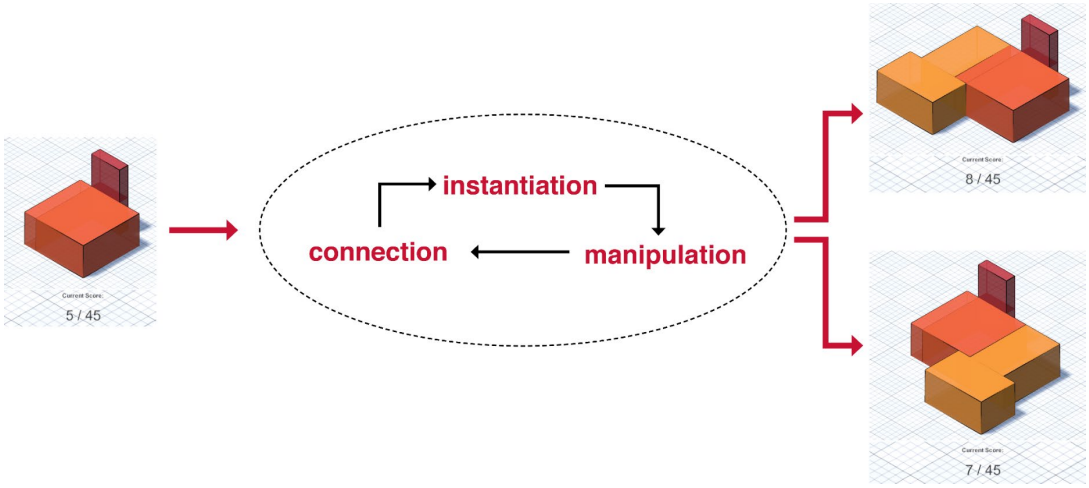


Figure 4-1 Original Action Cycle

Source: Author

The resulting action cycle was expanded upon during implementation as different types of player actions needed to be accounted for which fell outside arranging the shapes one after another. Another way of looking at it is that the action cycle is comprised of the types of actions available to the player based on their previous action.

While the overall action cycle has grown in complexity as more and more features were added, the core action cycle, the action cycle consisting of the very basic actions a player takes, has stayed virtually unchanged since deciding on a scenario. This core action cycle serves as the backbone for all gameplay where each puzzle piece the player interacts with has its own action cycle comprised of three basic player actions: *instantiation, manipulation,* and *connection.*

Each action cycle starts *instantiation* and ends with *connection*, after which the game provides the player with feedback on the choices they made during in the action cycle as and saves those choices as a *design rule* within a *design schema*.

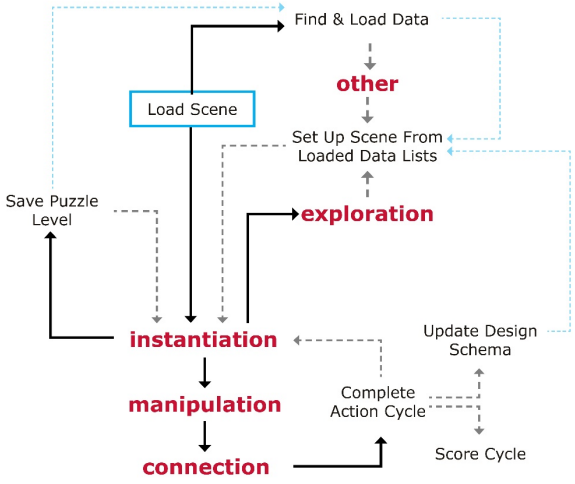


Figure 4-3 Simplified Action Cycle

Source: Author

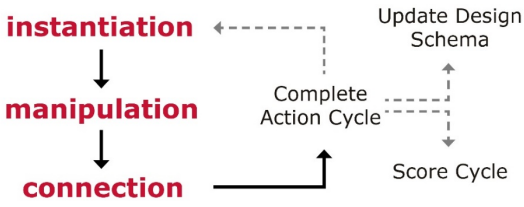


Figure 4-2 Core Action Cycle

Source: Author

Later sections in this chapter will explore what it is that happens at each step in this cycle as well as how they impact the overall player experience and gameplay difficulty.

4.4 Instantiation

The *instantiation* action cycle state acts both the point of entry and exit for the core action cycle. It is the default state whenever a player first loads a puzzle level or if they load a previous save. From the *instantiation* state, a player can do the following:

- select an object to *instantiate* and progress to the *manipulation* state (section 4.6 Manipulation)
- save their current progress.
- enter to the *exploration* action cycle state (section 4.10 Exploration)
- undo previous action cycle (section 4.9 Undo)

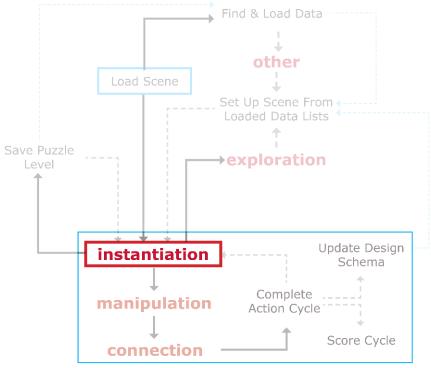


Figure 4-4 Instantiation's Location in Simplified Action Cycle

Source: Author

During the *instantiation* state of the action cycle the player selects a puzzle piece to be placed

within the scene. The puzzle piece they select at this point is the object the player will be acting upon within this action cycle, i.e. the instantiated puzzle piece becomes the selected puzzle piece unless it is the first or only puzzle piece instantiated in the scene, in which case it becomes the *player initial object*. The player selects the puzzle piece for instantiation by clicking on a button corresponding to the desired piece located at the top of the game screen.

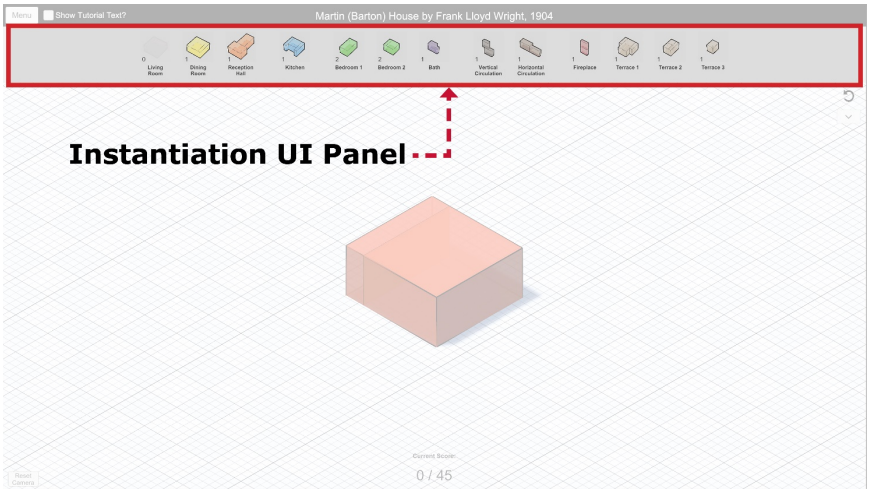


Figure 4-5 Instantiation UI Panel Location

Source: Author

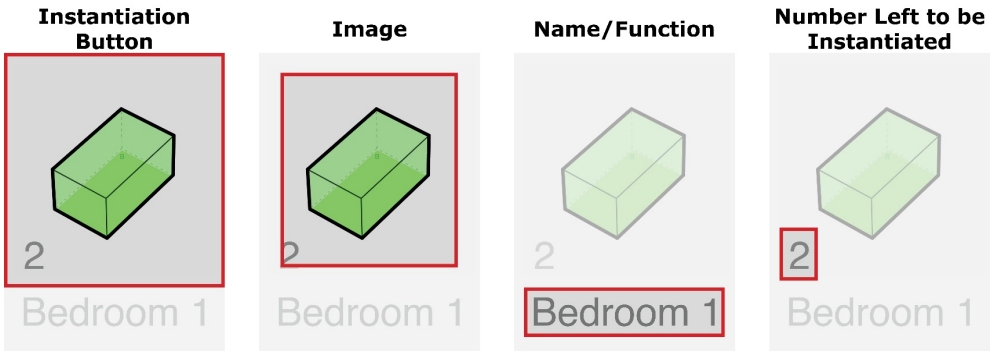


Figure 4-6 Instantiation Button Labeled Example

Source: Author

Each of these buttons corresponds to a puzzle piece prefab in the set puzzle pieces the player is provided. It is important to note that the set of puzzle pieces can include multiple instances of a single puzzle piece prefab, as is the case with both bedroom 1 and bedroom 2 of the Martin House puzzle (see page 197 for the set puzzle piece prefabs for the Martin House puzzle). The buttons are designed to graphically convey information about the puzzle pieces in the set. This information

currently includes an image of the puzzle piece that would be instantiated if the player clicks on it, text telling the player what that puzzle pieces function/name is, and text telling the player how many of that particular puzzle piece prefab are left in the provided set that still need to be placed within the scene. The image of the puzzle piece is currently not of the prefab itself, but a drawn representation made in Illustrator to help the overall legibility of the linework distinguishing the basic shapes that comprise the overall puzzle piece.

Since the player is instantiating puzzle pieces from a finite set, the game needs to know when to stop instantiating a particular puzzle piece prefab and communicate that information to the player. The game does this by disabling the buttons when necessary so that clicking on them does not do anything and conveying this fact by fading/graying out the image assigned to the button. There are three cases where the game has to do this. The first is when there are no puzzle pieces currently within the scene (like when the puzzle level is loaded for the first time for example), where only puzzle pieces located on the ground floor are able to be implemented. The second case is when all instances of a puzzle piece prefab in the provided set have been instantiated. This lets the player know at a quick glance how close they are to completing a level and what pieces they have left to work with. The third case is when the player already has a selected puzzle piece within the scene, i.e. they are outside the *instantiation* action cycle state.

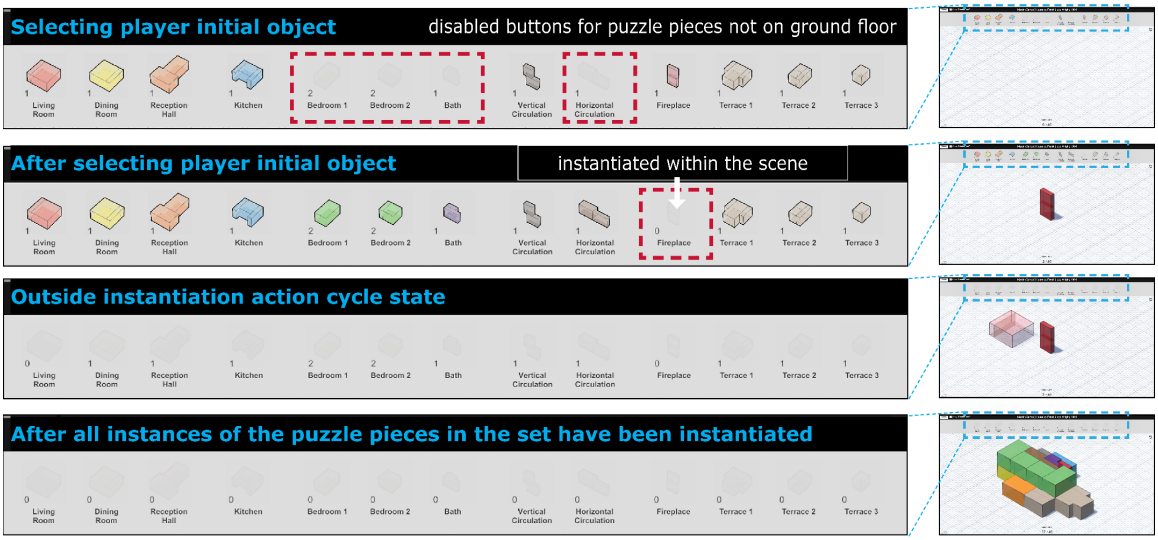


Figure 4-7 UI for Instantiation – Enabling & Disabling Button

Source: Author

4.5 Manipulation

Once a player has selected a puzzle piece to be placed/instantiated within the scene, they then have to determine where within the scene they want it to go. This stage of the action cycle is *manipulation*, as the player is manipulating the puzzle pieces location within the game space. The *manipulation* state can only be entered from the *instantiation* and *connection* states, depending on if the player is progressing forwards or backwards through the player action cycle. From the *manipulation* state a player can progress to the *connection* and *instantiation* states, also depending on if the player is progressing forwards or backwards through the player action cycle. To progress forward through the player action cycle to the *connection* state, the player can use either the 'enter' key on their keyboard or by clicking on the current selected puzzle piece with their left mouse button.

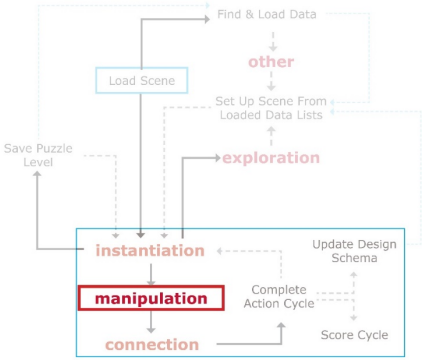


Figure 4-8 Manipulation's Location in Simplified Action Cycle
Source: Author

Currently *manipulation* involves the translation and rotation of the puzzle piece, however, during the initial development stages of this project several different geometric transformations where discussed. These transformations included the three Euclidian geometric transformations: translation (changes the position), rotation, and reflection; and one non-Euclidian transformation: scale. Based on the initial testing that had been done with *Unity* detailed in section 2.1, having the player be able to adjust all three was possible. However, we decided to drop scaling and reflection because they are not useful tools for the player based on the current concept for this game and would only make things much more difficult if not impossible for a player.

Simply stating that the player is able to manipulate a puzzle piece by transforming its position and rotation is not enough however. How that transformation happens specifically is important. Both transformations had already been explored in the prior *Unity* experiments detailed in section 2.1, where there seemed to be two key ways of transforming a puzzle piece's position (translation) and rotation. These transformations could either be smooth or happen at set increments. There are drawbacks to both methods, especially when it comes to transforming a puzzle pieces position. If translation occurs smoothly the puzzles

within the game become extremely hard if not impossible to solve because, without constraints or being able to lock onto specific points of reference on the puzzle pieces themselves, it you cannot really accurately rearrange pieces within the game space. On the other hand, if translation occurs using fixed increments that is maybe also not ideal as it can be difficulty to find an increment that will work for every puzzle piece that is not so small it might as well be smooth. Regarding rotation, it does not have the same drawbacks with using fixed increments that translation did, so rotation currently happens at a fixed increment determined by the precedent, in the case of the Martin House puzzle this increment is 90°.

So, adapting rotation from the prior work done in *Unity* was not a problem, but that still left translation unsolved. To rectify this, other methods of adjust the position where looked at, specifically what constraints could be used to help smooth translation be more accurate. Currently, any manipulation of a puzzle piece's position is constrained by object snapping where the only way for a player to progress forward in the action cycle to the *connection* state the player needs to have their selected puzzle piece be snapped to another puzzle piece within the scene. The type of object snapping used in the current build of the game is corner-to-corner snapping where, as the player is moving the selected puzzle piece around, if one of its corner gets within a certain distance of another puzzle piece's corner the selected piece moves so that those corners overlap.

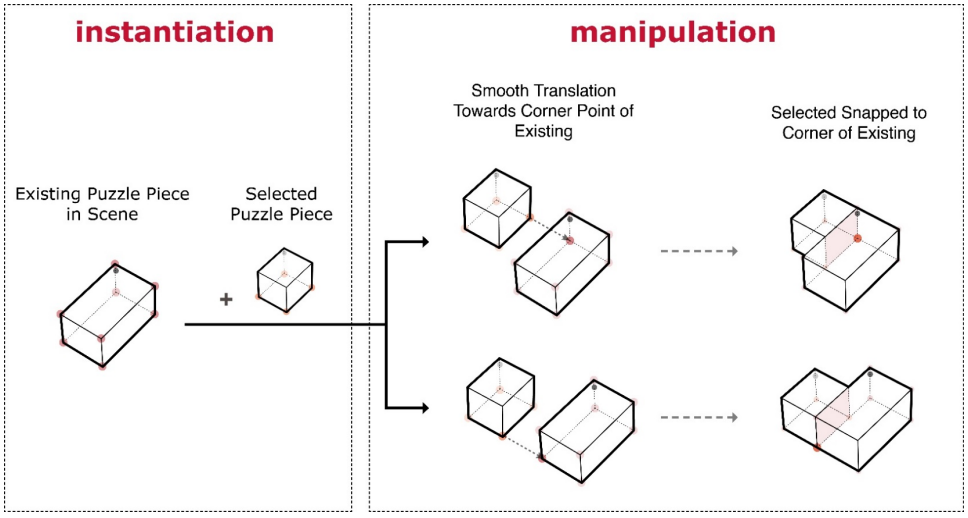


Figure 4-9 Proposed Solution for Smooth Translation: Corner-to-Corner Snapping
Source: Author

Now that how the selected puzzle piece behaves during the *manipulation* state is figured out, the next step is defining the player inputs for those behaviors. In figuring out the player inputs to be used for *manipulation*, inspiration was taken from the games I have played in the past that have some kind of build mode, like the Sims. In the prior *Unity* experiment the WASD keys were used to control the puzzle piece's position and the ',' and '.' keys were used for rotation. While the rotation inputs were kept from the previous *Unity* experiment, this was not the case for the position inputs. One of the findings from the initial puzzle experiments done in Rhino was that the player was going to need to have at least some control over the camera (see section 3.4.4). It was decided to change the position input from the WASD keys to being based off the player's mouse position on the screen. This both feels more natural during gameplay and frees up the WASD keys to be used for controlling camera movement.

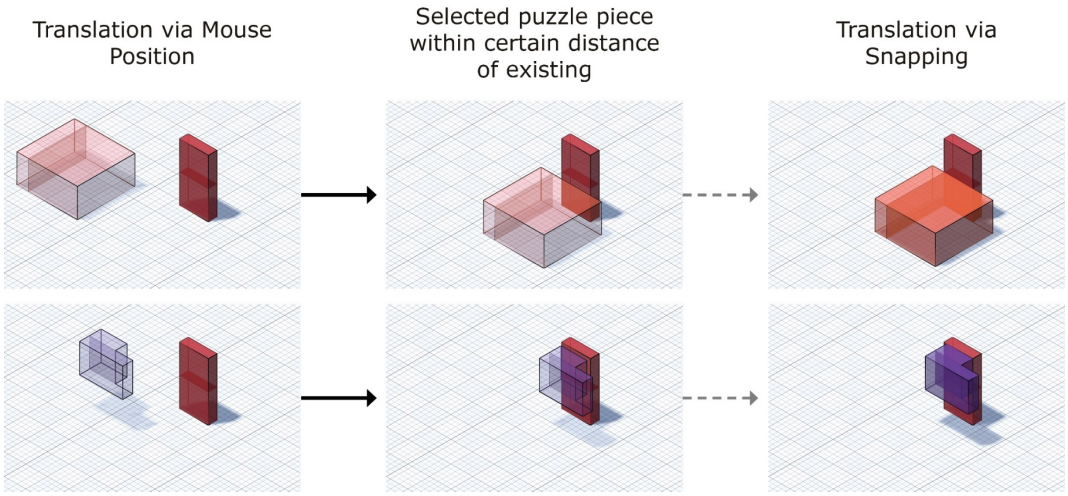


Figure 4-10 Translation via Mouse Position with Corner-to-Corner Snapping

Source: Author

Once a player has placed the selected puzzle piece where they want, so long as it has been snapped to another puzzle piece in the scene, the player can progress to the next state in the action cycle by either pressing the 'enter' key or by left clicking with their mouse.

4.6 Connection

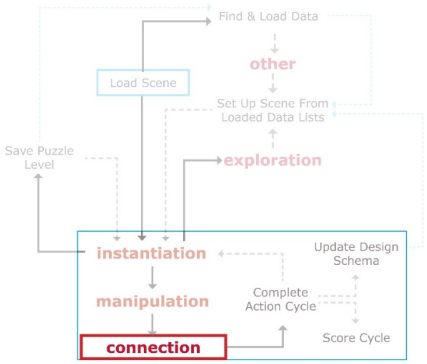


Figure 4-11 Connection's Location in Simplified Action Cycle
Source: Author

While in the *connection* action cycle state the player defines the circulation connections between the current selected puzzle piece and any adjacent puzzle pieces. *Connection* is the last action cycle state of the core action cycle and can only be entered from the *manipulation* state. To exit the *connection* state the player can either progress forwards in the action cycle, completing it and starting a new action cycle, or progress backwards and return to the *manipulation* state. To exit the *connection* action cycle state, the player can either press the 'enter' key on their keyboard or click on the check button located at the top middle of their screen beneath the instantiation UI panel. The player can choose to complete the action cycle, exiting the *connection* state, regardless of whether or not they have actually connected any puzzle pieces.

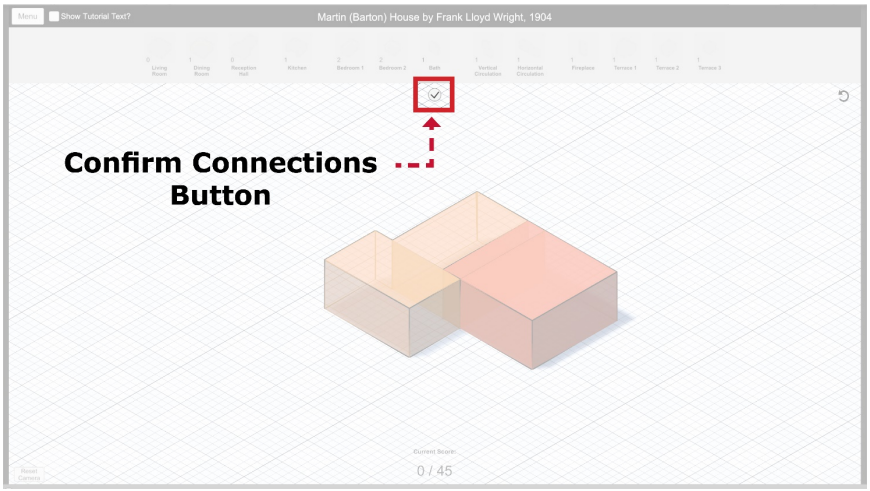


Figure 4-12 Confirm Connection Button
Source: Author

This action cycle state helps provide more architectural meaning to the puzzles as the connections can be seen as representations of architectural elements like doors or archways and helps to define how different room functions are connected to one another. This action cycle state is currently the most complicated of the core action cycle state, requiring the automatic set up of puzzle pieces already

placed within the scene during the *manipulation* state prior to entering the *connection* state. However, in terms of gameplay, the *connection* state is fairly straight forward. Within the *connection* state the player is able to click on any of the puzzle pieces in the scene adjacent to the current selected puzzle piece to toggle whether or not it is "connected" to the selected puzzle piece or not.

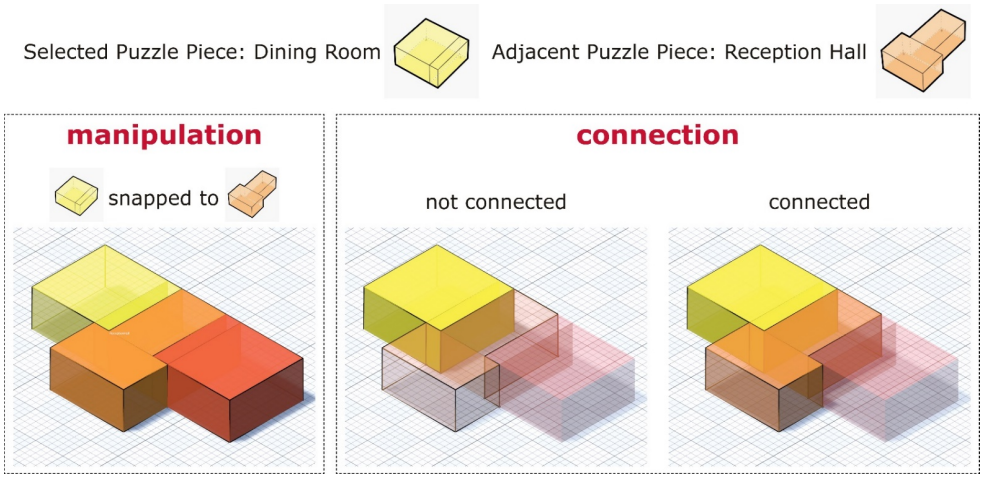


Figure 4-13 Toggling Connection Example

Source: Author

4.7 Scoring

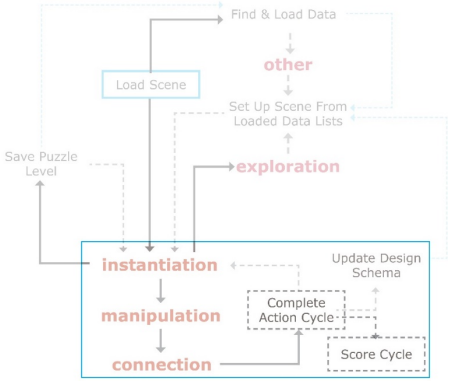


Figure 4-14 Scoring's Location in Simplified Action Cycle

Source: Author

Scoring is used as a means of providing the player with the meaningful feedback required in the attainment of tacit knowledge. Feedback is an integral part of video games and is at core of the relationship between video game and player. Making sure to provide the player with meaningful feedback provides them with tools and information to help guide them towards a desired goal, however, placing too much emphasis on feedback can negatively impact player experience as well.[4] Getting the balance right where the game can provide a player with meaningful feedback without overwhelming them is a tricky task that this project does not have time to full investigate. Instead, this project uses a basic preliminary scoring system is used as a

starting point for what will hopefully be a more robust and subtle scoring system in the future.

One of the reasons behind the selection of Scenario 3.1 back in Chapter 2 was the potential to use existing architectural precedents as a means of generating a series of criteria that can be used to provide player's with feedback through a scoring system. Scoring is one of the automatic actions done by the game when a player completes an action cycle just prior to the start of the next *instantiation* state. This is done even the player has not gone through a complete core action cycle (as is the case when designating a *player initial object*).

Scoring looks at two different criteria relating to the player's faithfulness to the original precedent's design: form/spatial arrangement and relationship between functional zones. Faithfulness to the original design's form/spatial arrangement measures how close a player got to arriving at the same arrangement as the abstracted original. The more shapes that match the positions and rotations found in the original, the higher the player's score.

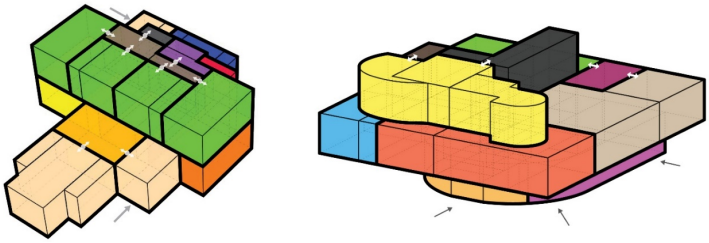


Figure 4-15 Original Arrangement for Martin House (left) & Villa Savoye (right)
Source: Author

Faithfulness to the original design's relationship between functional zones measures how accurate the functional connections made by the player during gameplay are when compared to the functional connections found in the original. The player's score increases for each correct function connection they made.

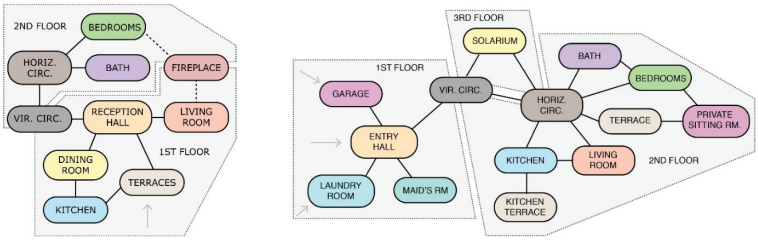


Figure 4-16 Original Functional Connections for Martin House (left) & Villa Savoye (right)
Source: Author

A player's score is tracked as a numerical value. The player's score increases for every one of the criteria they have met from the two criteria lists, one for the design arrangement and one for the functional connections. If a player deletes/destroys a puzzle piece for whatever reason, if the decisions made during the puzzle pieces action cycle increased the player's score, their score is reduced by the same amount. The score is then displayed at the bottom center of the players' screen.

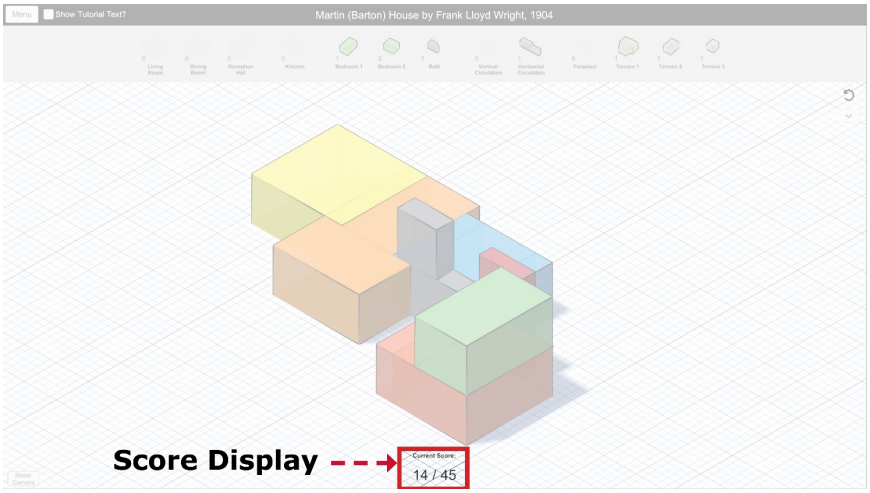


Figure 4-17 Score Display Location

Source: Author

This scoring system is not meant to be the final version and is very much in its infancy but hopefully it acts as a step towards being able to provide meaningful feedback required for tacit knowledge to be gained by the player.

4.8 Design Rules and Schema

The *design schema* is used to track player decisions which are saved as *design rules*. The *design* schema is updated right after the player's score has been updated and is one of the automatic actions done by the game when a player completes an action cycle just prior to the start of the next *instantiation* state. This is done

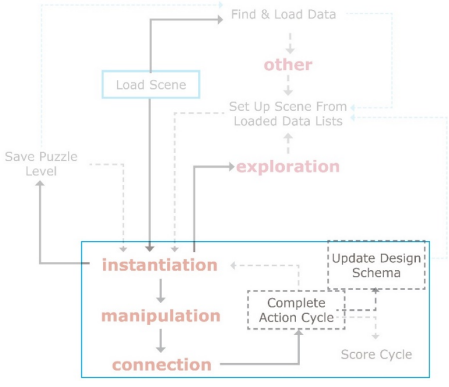


Figure 4-18 Schema Update's Location in Simplified Action Cycle

Source: Author

even the player has not gone through a complete core action cycle (as is the case when designating a *player initial object*).

Design rules contain information about the shape, function, position, rotation, and functional connections about an action-cycles. The *design schema* reflects the order in which the player arranged the puzzle pieces in the scene and any backtracking they may have done. This information is used for a few things on the player end. It primarily used to allow the players to go back to a previous step in their process and to go back and explore any alternative solutions or partial solutions they have created during this process. The former relates to the undo action described in section 4.9 and the later forms the *exploration* action cycle state examined in section 4.11.

4.8.1 Design Rules

A design rule contains information about the choices a player made within an action cycle. This information includes details about the puzzle piece, its position and rotation in the scene, a list of its functional connections, and whether or not any of the scoring criteria were met. The specifics of what exactly is stored in a design rule is detailed in section 5.8.1 in the chapter dealing with the actual implementation of the current game's build.

Once a new design rule is created it is saved to a list of design rules in the design schema list. Design rules are used to help support the undo action cycle player action and in the *exploration* action cycle state. In both instances the player is looking back at their past decisions.

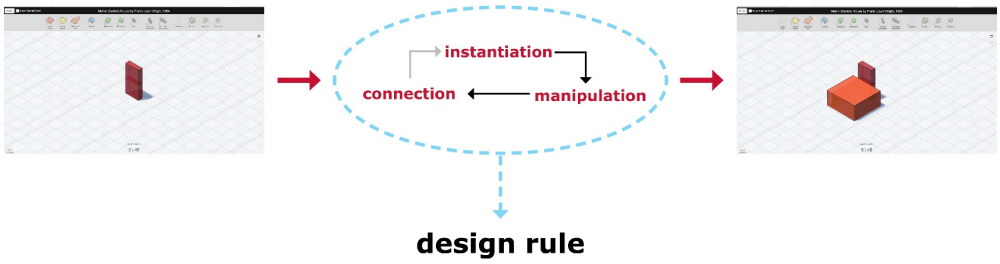


Figure 4-19 Design Rule from Action Cycle

Source: Author

4.8.2 Design Schema

The design schema is the compilation of design rules structured to reflect the various paths a player has taken within a puzzle, accounting for both the order in which puzzle pieces where arranged and any backtracking a player did. Using the diagrams created for iteration three of the Martin House, this idea of tracking *design rules* as the *schema* and what that might look like in terms of gameplay was interrogated. This idea of the branching paths a player can take to arrive at a solution including any backtracking they might have done, was interesting. The *design schema* is a way for the game to track a player's progression and the variations that result from any backtracking.

The following two pages provide an example of a *design schema* for the Martin House puzzle level.

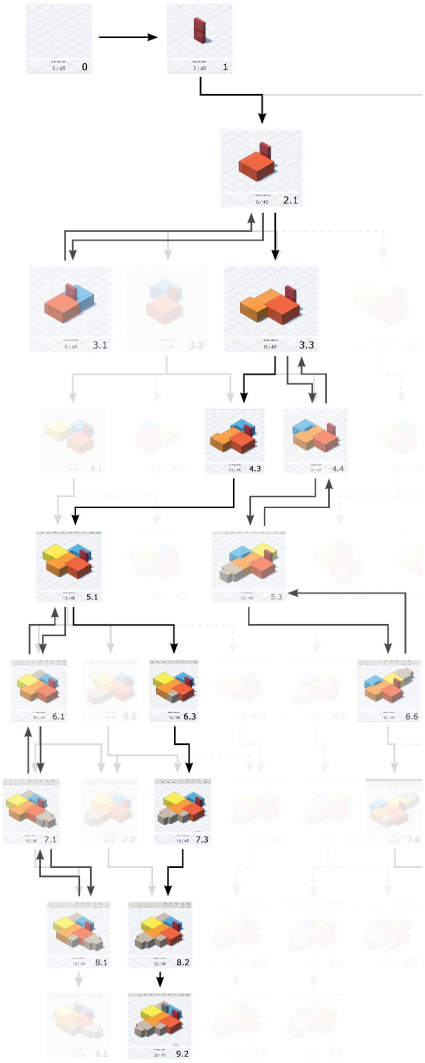


Figure 4-20 Variation through Progression & Backtracking
Source: Author

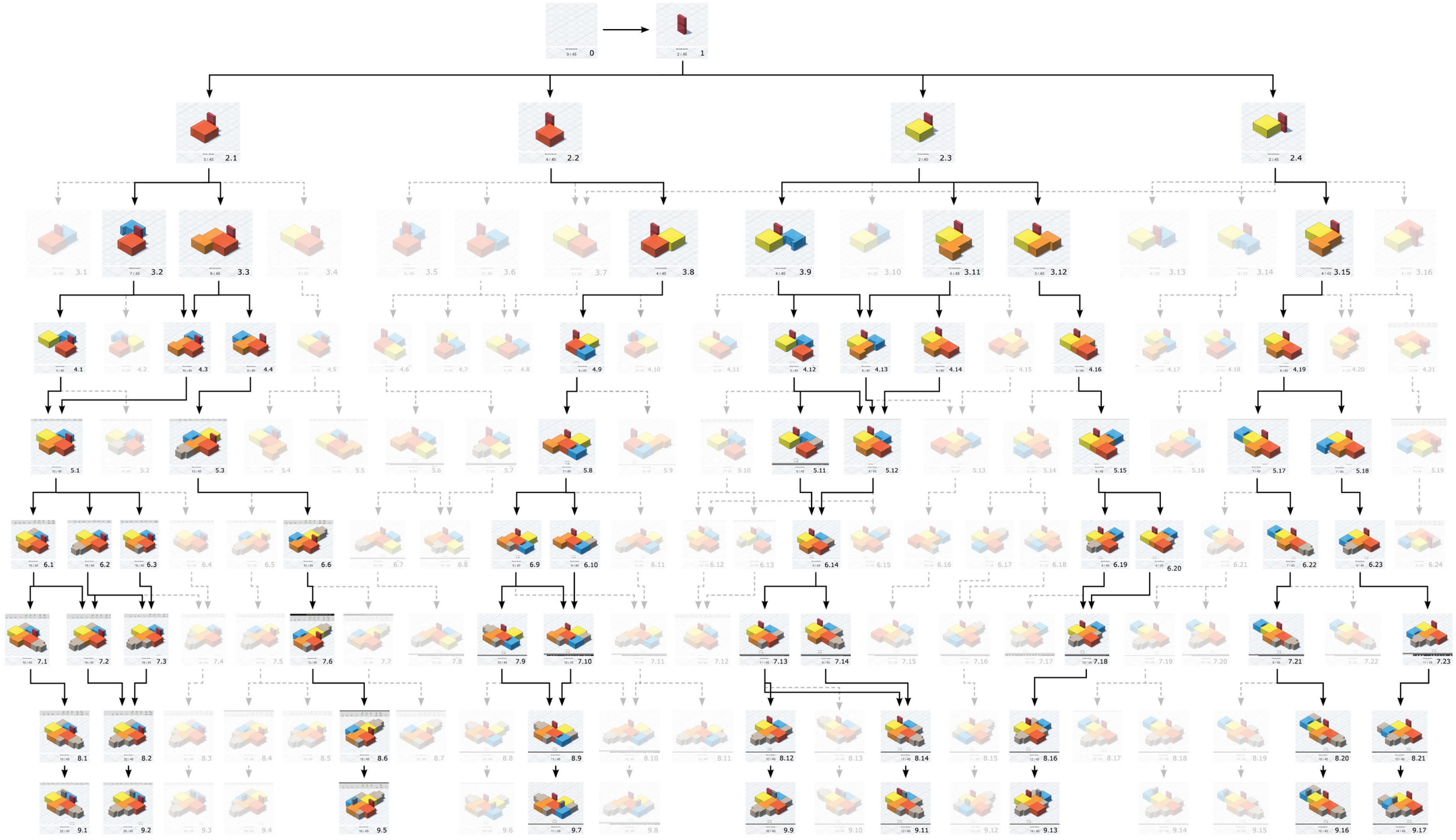


Figure 4-2 Design Schemata for the First Floor of the Martin House (Barton House)

Source: Author

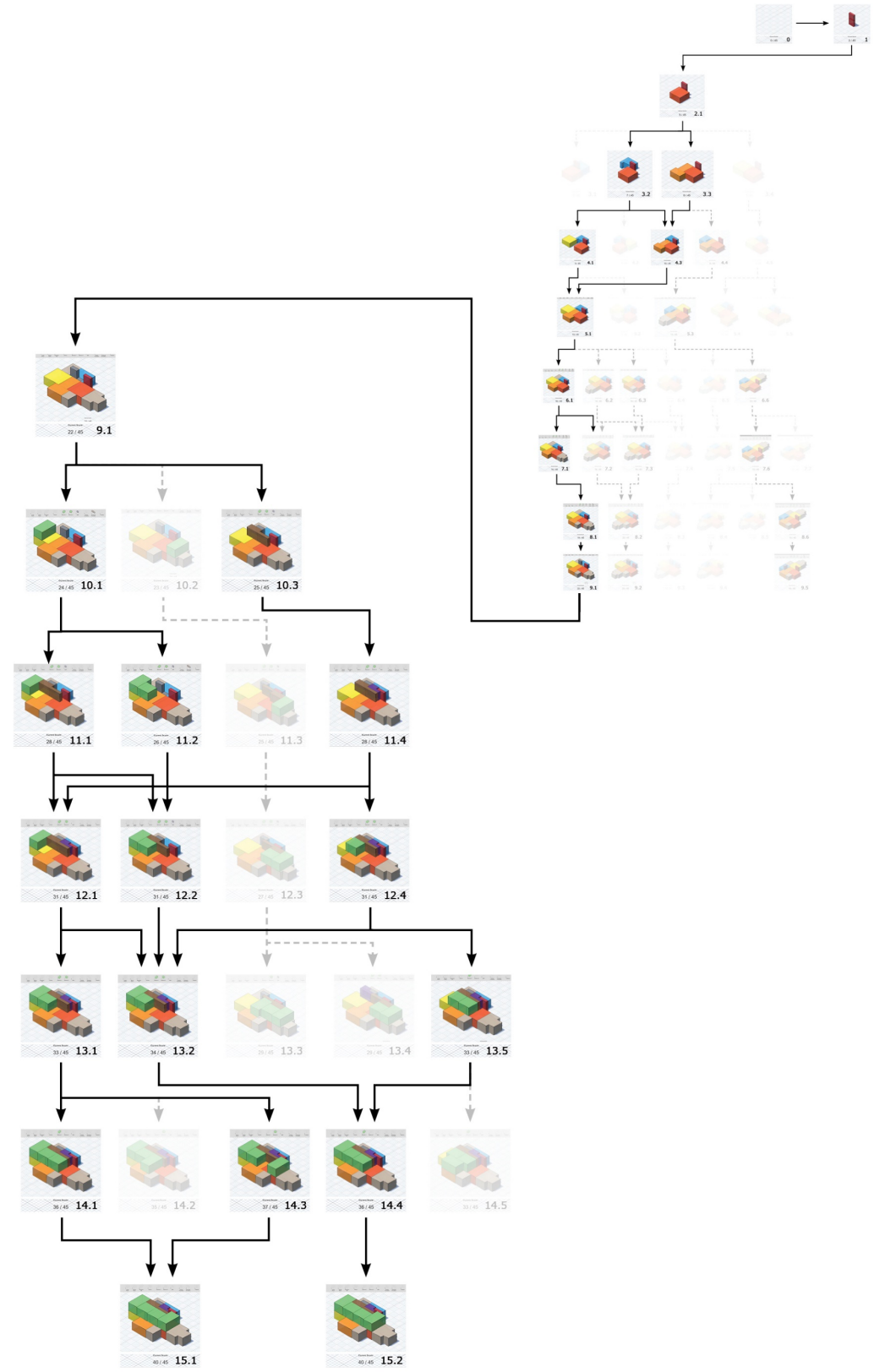


Figure 4-22 Design Schemata for the Second Floor of the Martin House (Barton House)

Source: Author

4.8.3 Design Variation

Not every player is going to prioritize obtaining a perfect score for each level, caring more about the freedom of form-making. The ability for players to create design variation becomes an important consideration when up each scene in the *Unity* editor. Several elements of a precedent's abstraction will affect how easily a player will be able to create a variation which makes sense using the vocabulary provided to them. Things like alignment and uniformity can help provide the player with options when arranging the puzzle pieces that make sense even if they are not accurate to the original precedent and without those options it becomes very difficult and frustrating to try and create variations. It is not just the abstraction itself that determines how difficult it is for a player to create alternative designs for a given precedent which make sense, but the precedent itself.

Precedents, like the Martin House, lend themselves very easily to the creation of design alternatives due to the alignments and uniform shape/placeholder dimensions in the layout's abstraction. This seems to be a result of the "organic" nature of the prairie style's design process.[5] Meanwhile, precedents like the Villa Savoye do not lend themselves easily to this process of design variation due to the sheer number of shape/placeholder sizes, commonality of unique shapes, and lack of meaningful alignment. This seems to be a result of the process by which it was designed, what seems to be a more outside to inside approach.

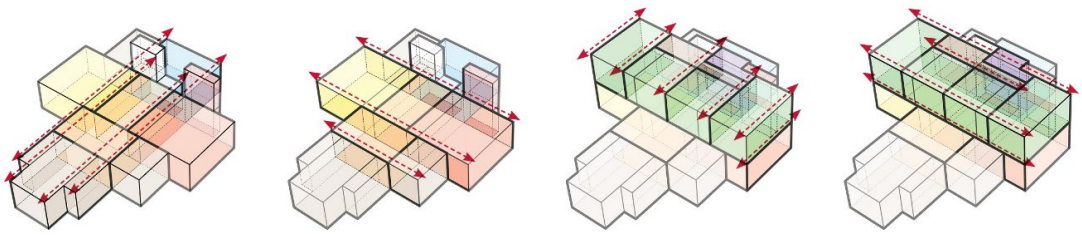


Figure 4-23 Alignment in the Design of the Martin House Iteration 3

Source: Author

Within the abstraction process of the Martin House, and the preliminary exploration of how the specifics of the abstraction process would affect the players capacity to generate design variations that make sense, specifically in terms of the Martin House, I found an interesting parallel between my own explorations of Martin House puzzle solution variations and the shape grammar for the Prairie Style detailed by Koning and Eizenberg.

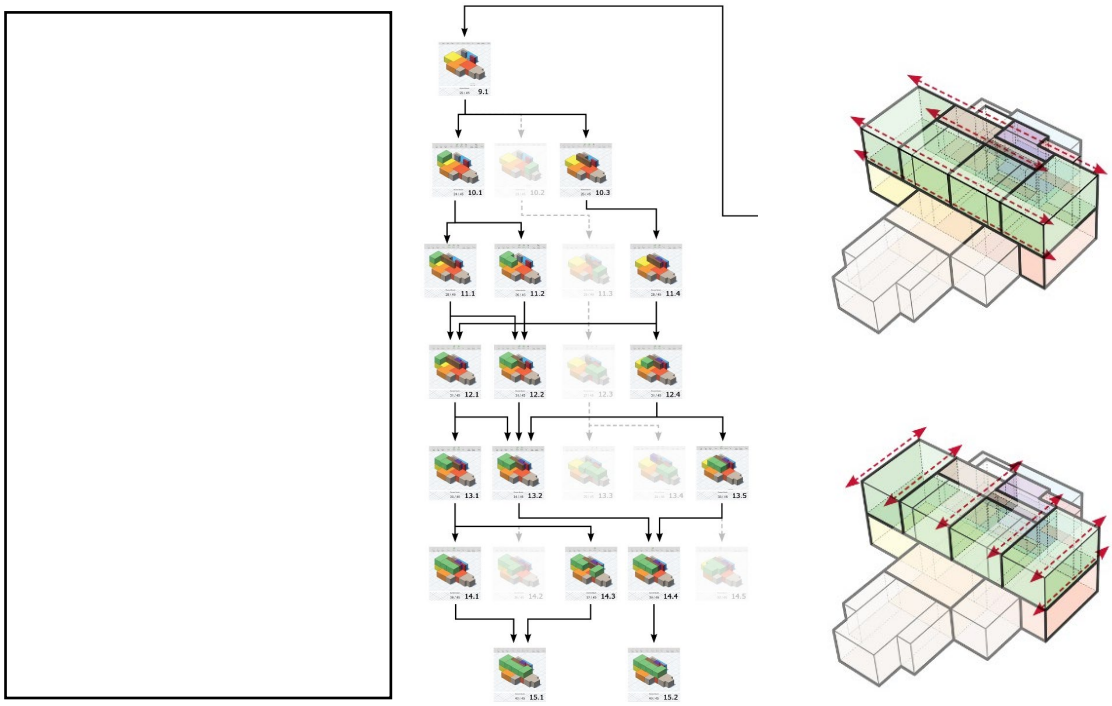


Figure 4-24 Second Floor Creation using Design Rules (left) Compared to Second Floor Creation Using Puzzle Pieces (center) & Alignment of Abstracted (right)
Source: Koning and Eizenberg (left) [6] & Author (center & right)

4.9 Undo

Based on initial experimentation done using the Martin House puzzle pieces done both in Rhino and *Unity* it became abundantly apparent that the game would need to be capable of providing the player with a means of backing out of a decision or choice. The current iteration of the game has two kinds of undo both relating to specific parts of the core action cycle.

The first of these two times is when a previous action cycle state is undone, undo within an action cycle. As the *instantiation* state serves as the entry point into the action cycle, it is the only state of the three core action cycle states to not allow the player to perform this type of undo. This involves moving one step backwards within the cycle, deleting or otherwise returning the

Figure 4-25 Undo within Action Cycle
Source: Author

scene to what it was just prior to the player exiting the previous action cycle state before then re-entering said state. So, if a player was in the *connection* action cycle state they would go to the *manipulation* action cycle state, having all the connections they formed up till then removed and allowing the player to once again move their *selected* puzzle piece around. If the player is in the *Manipulation* action cycle state then their selected puzzle piece is simply destroyed and the player is taken back to the *instantiation* action cycle state.

The second type of undo is when the player undoes a previous action cycle. In this type of player undo, the player backtracks, reversing any decisions made in the most recent *design rule.*

The player can perform the undo action in a few different ways, depending on the type of undo they are doing. In the top right corner of the screen is an undo button that will call either of the two times of undo depending on where in the player action cycle they are. If the game is currently in either the *manipulation* or *connection* action cycle state (i.e. the player is wanting to undo within the current action cycle), then the player is also able to perform the undo action by pressing the 'escape' key on the keyboard.

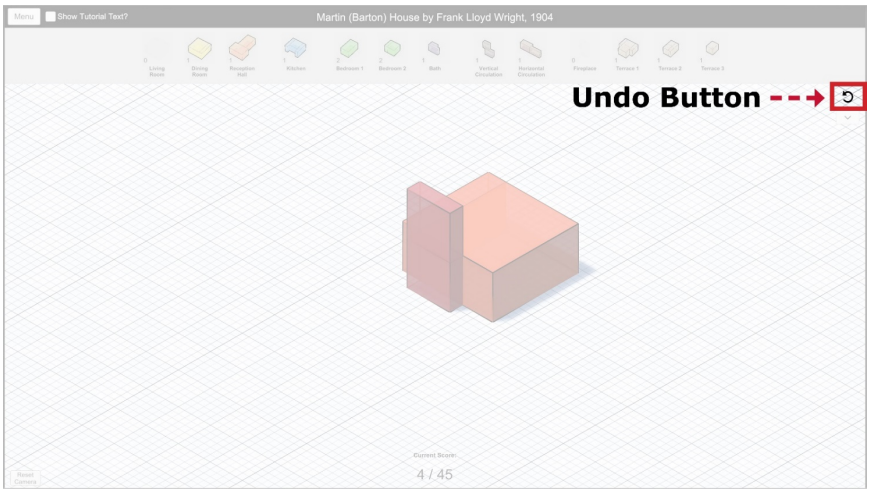


Figure 4-26 Undo Button Location

Source: Author

4.10 Exploration

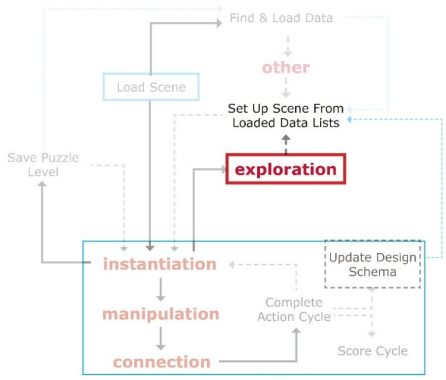


Figure 4-27 Exploration's Location in Simplified Action Cycle
Source: Author

Exploration is similar in nature to the undo action, however, it is more sophisticated but limited in its application. The *exploration* action cycle state can only be entered from the *instantiation* state and always exits to the *instantiation* state. The player enters the *exploration* state by clicking on a down arrow button located just beneath the undo button.

Currently the player is only able to enter the *exploration* state from the *instantiation* state. The button however, does not become disabled when the player isn't in the *instantiation* state. Instead, if the player attempts to enter the *exploration* action cycle state while in either the *manipulation* or *connection* states, the game will perform the undo within action cycle action, once for *manipulation*, and twice for *connection*, returning the player to the *instantiation* action cycle state before allowing the player to do any exploration.

They can then navigate to a place within the schema for that level using the *exploration UI sliders*. The game will set up the scene so that it shows the player what puzzle pieces would need to be added or subtracted form the scene if they wanted to go back to that particular spot in the schema. The player can then chose to exit out of the *exploration* action cycle state by clicking on the same button to hide the *exploration UI* before returning to the *instantiation* action cycle state.

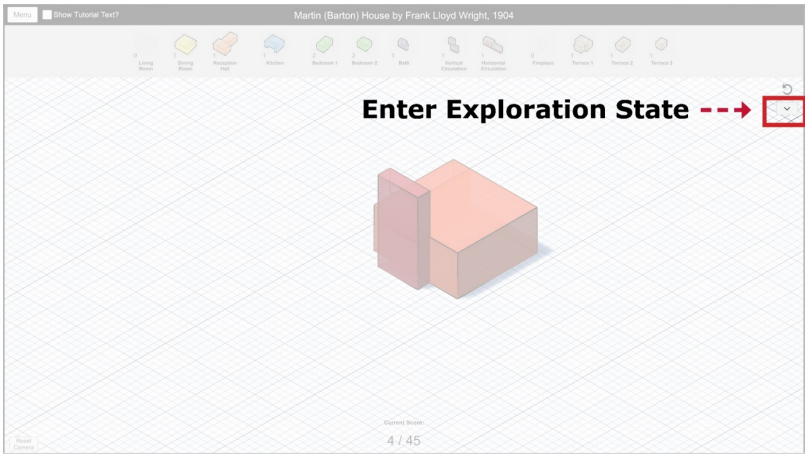


Figure 4-28 Enter Exploration State Button Location
Source: Author

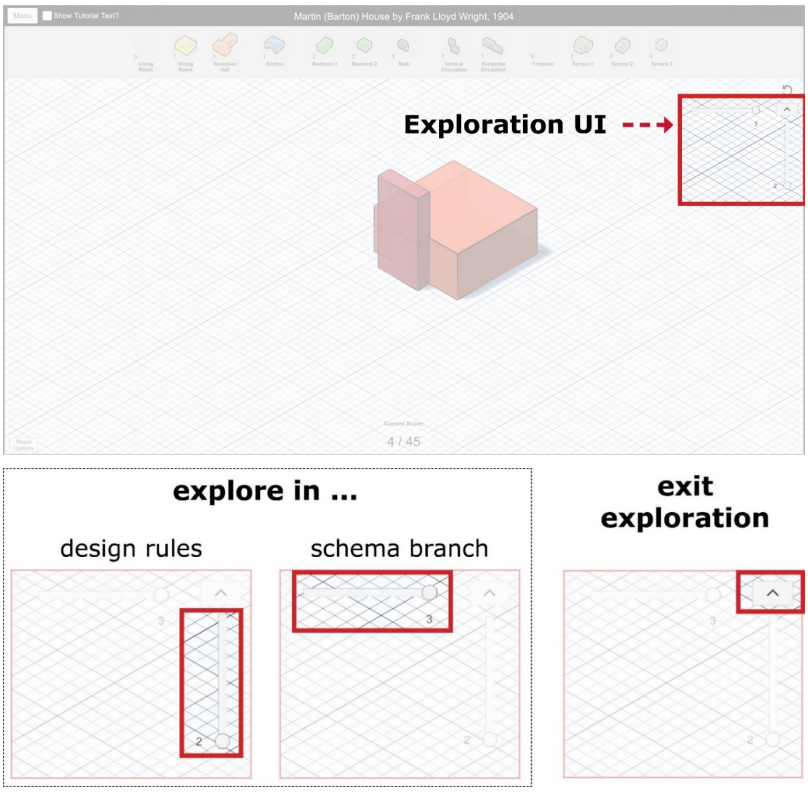


Figure 4-29 Exploration UI Location & Layout

Source: Author

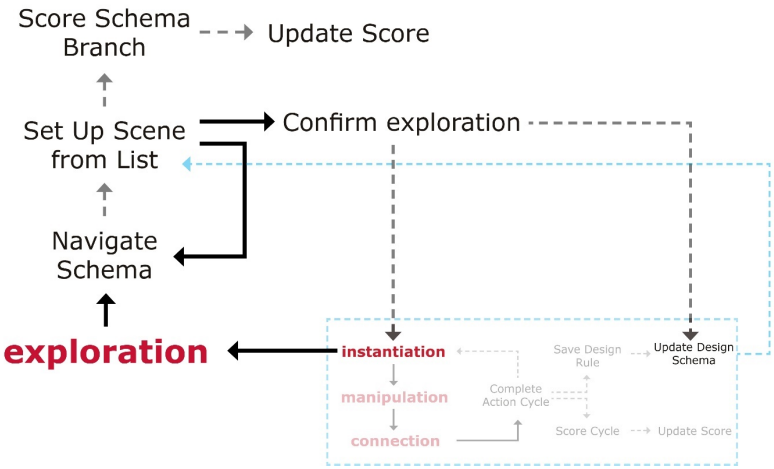


Figure 4-30 Exploration Action Cycle

Source: Author

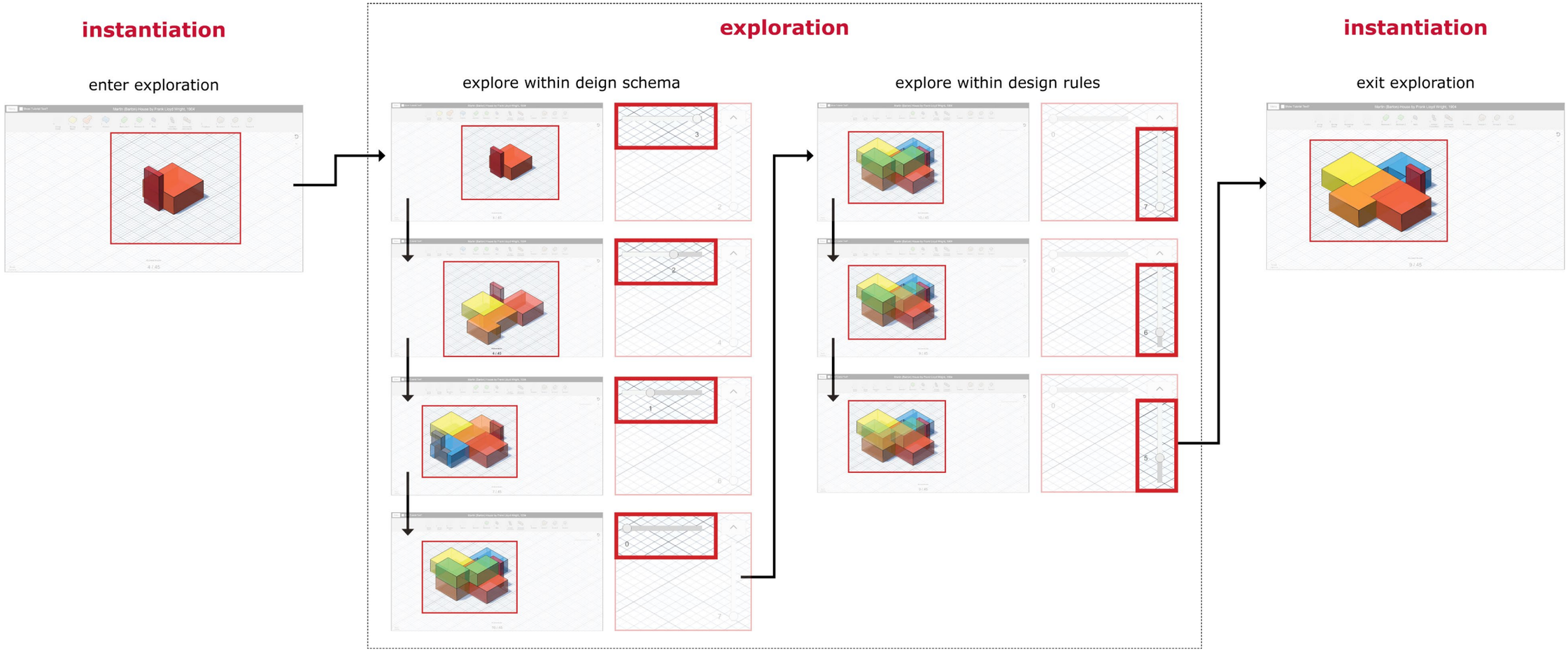


Figure 4-3 Design Schemata for the First Floor of the Martin House (Barton House)
Source: Author

4.11 Difficulty

Something that became apparent early on in the abstraction process was how minor changes to shapes and the constraints applied to the actions players do to those shapes greatly effects how difficult a particular puzzle is to solve, some of which were looked at throughout chapter 3. These gameplay elements can be put into two overarching categories:

1. Information provided to the player
2. Constraints placed on player actions

The sections will go through the specifics of each of the two overarching categories used in this project when thinking about player difficulty.

4.11.1 Information Provided to Player

One of the key issues that came up early in this project was the question of how much information to provide the player at the start of the level. If too much information is given, then the puzzle is not really much of a puzzle, but if not enough than the player comes in directionless in what could easily end up as a deeply frustrating endeavor on their part.

Types of information provided to the player have been thought of in three different ways:

1. *Embedded Information.* Information embedded within the puzzle pieces themselves, like having them already be labeled when the player starts the level.
2. *Contextual Information.* Images and information about the precedent shown to the player either prior to them going through process of solving said precedent's puzzle.
3. *In-Game Hints.* Information contained within the scene which could help guide players if they get lost.

Between these three types of information most of the attention has been given to what information has been embedded within the puzzle pieces themselves. *Embedded information* refers to information conveyed through the core parts of the

game that the player is interacting with during the core action cycle. It includes information about some of the formal aspects about the puzzle piece and can be affected by things like whether or not compound shapes come-pre-grouped or if puzzle pieces are assigned to a specific floor, or does the player have to set that.

This type of information can also pertain to things about the functional aspects of the puzzle pieces and is affected by things like whether or not the player has to assign function to the various puzzle pieces (the *naming* part of earlier action cycles explored early on in the project) or do they come pre-labeled.

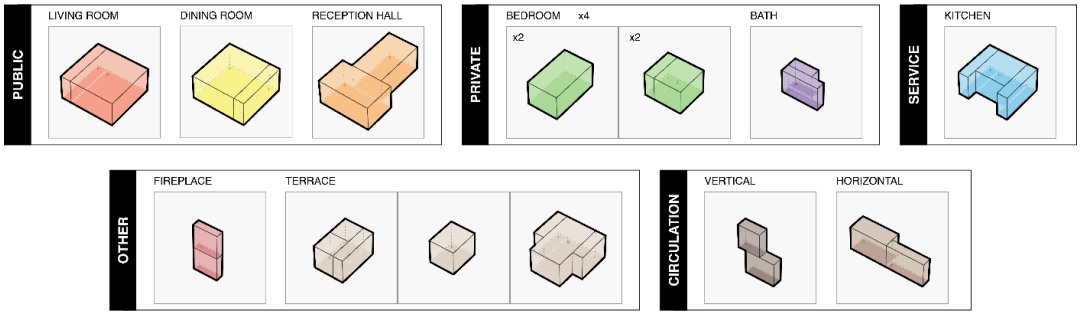


Figure 4-32 Example of Pre-Labeled Shapes

Source: Author

4.11.2 Constraints on Player Actions

The other thing that affects gameplay difficulty is the amount of freedom a player has within the game. What can and can't they do within the confines of the game. While restrictions on player agency exists throughout the game due to the nature of how it was implemented, some additional constraints were added. These constraints were all detailed in their respective action cycle state's section. For *instantiation* this constraint consists of limiting the number and order a player can instantiate objects within the scene. For *manipulation* the constraints take the form of corner-to-corner snapping which limits player translation of a *selected* puzzle piece and limitation of player rotation.

[1] Park and Vakaló, "A Form-making Algorithm. Shape Grammar Reversed."

[2] John M. Carroll, "Introduction: The Scenario Perspective on System Development," in *Scenario-based design : envisioning work and technology in systems development*

(New York: New York : Wiley, 1995).

[3] Dave Westwood and Mark D. Griffiths, "The Role of Structural Characteristics in Video-Game Play Motivation: A Q-Methodology Study," *Cyberpsychology, Behavior, and Social Networking* 13, no. 5 (2010), https://doi.org/10.1089/cyber.2009.0361.

[4] Ryan Rogers, "The motivational pull of video game feedback, rules, and social interaction: Another self-determination theory approach," *Computers in Human Behavior* 73 (2017), https://doi.org/10.1016/j.chb.2017.03.048.

[5] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."; Wright, *Frank Lloyd Wright: the early work*.

[6] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

Chapter 5: Implementation

The previous chapter outlined the basic gameplay features for a 3D puzzles game where players are given puzzles to solve that have created based off some architectural precedent. This chapter catalogs the development of a prototype build of one puzzle level derived from one precedent, the Martin (Barton) House by Frank Lloyd Wright. As mentioned in the introductory sections of this dissertation, this game is being developed using the *Unity* engine with all scripting done in C#.

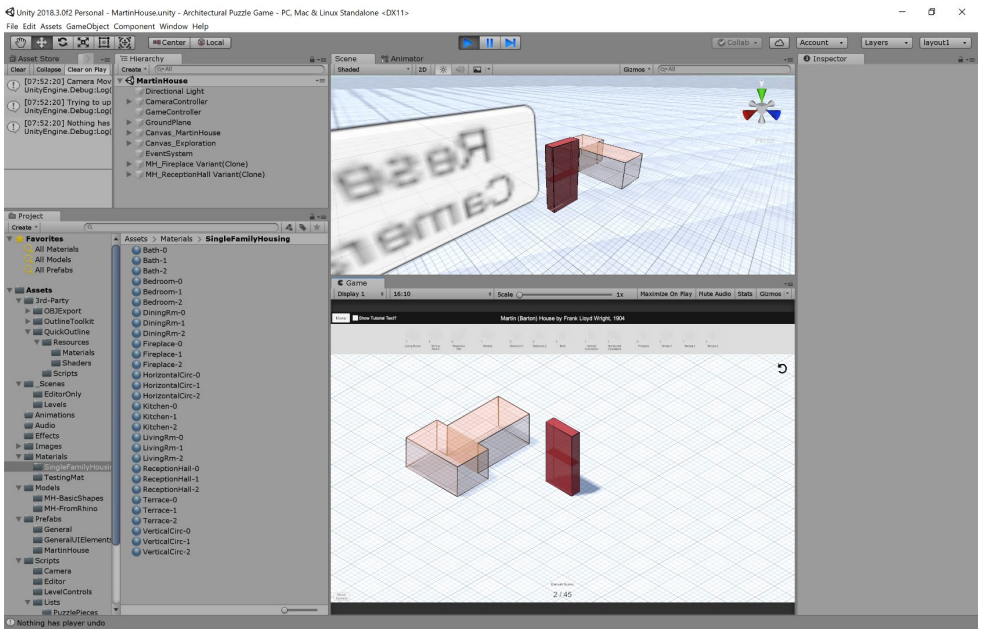


Figure 5-1 Screenshot of Current Build of Game in Unity Editor

Source: Author

The following features have been implemented in the prototype build created for this dissertation project:

- A set of *puzzle piece game objects* based off the Martin House
- Player *action cycle* in which a player *instantiates, manipulates*, and *connects* of a set of puzzle pieces belonging to a puzzle level
- The saving of *design rules* based off player actions and the generation of *design schema* comprised of those *design rules*
- Player *undo* individual actions within the current *action cycle* and an entire *action cycle*
- Player *exploration* of *design schema* within the current level

- *Scoring* player actions based on formal and functional criteria
- Basic camera controls, allowing the player to pan, zoom, and rotate their view of the game scene.
- Saving player's progress in a given puzzle level and then loading that save during a different play session.
- A basic main menu and in-game menu allowing the player to transition between scenes with infrastructure in place to allow for additional features.

The sections in this chapter will cover the implementation process for each of the above features, explaining how this iteration of the game was put together. This chapter will use a lot of *Unity's* terminology, refer to section 1.4.2 for to see their definitions.

Section 5.1 outlines how the Martin House puzzle level scene has been set up in the *Unity editor* and section 5.2 outlines how the puzzle piece game objects and prefabs were put together including their various parts. Section 5.3 goes over the action cycle and puzzle piece states. Sections 5.4 - 5.6 details how the core action cycle states, *instantiation, manipulation,* and *connection* were implemented with section 5.7 and section 5.8 going over the steps around the completion of a core action cycle, *scoring* and *design schema.* Section 5.9 and section 5.10 goes over the implementation of two means of backtracking made available to the player, *undo* and *exploration.* Finally, section 5.11 outlines the implementation of camera controls available to the player.

5.1 Scene Set Up

Each puzzle scene contains several game objects. There is a camera and a light so the player can see things in the scene, a ground plane, a UI for players to interact with, and a game controller which, as the name would suggest, controls the scene. The game controller is an empty game object that holds several different scripting components necessary for the game to run. In total there are six different scripting components, of which four are necessary for the game to run as intended, one is optional, and one is just for setting puzzle pieces in the editor. These six scripting components are:

- *Game Controller.* A custom class that handles action cycle states and stores the design schema. Required for gameplay.
- *Scoring.* A custom class that handles scoring player actions. Required for gameplay.
- *In Game Menu.* A custom class that handles the in-game menu UI and the behaviors and methods a player would call from that menu, like saving and loading. Required for gameplay.
- *Instantiate UI Manager.* A custom class that handles the UI used in the *instantiation* action cycle state, stores the *instantiation dictionary*. Required for gameplay.
- *Tutorial Text.* A custom class that updates any additional text that pops up during gameplay for the purposes of communicating what player inputs the game is looking for and what those inputs would do. Not required for gameplay, optional.
- *Puzzle Piece Material Manager.* A custom class used when setting up the scene in the editor to quickly make changes to material transparencies, not for normal gameplay.

The following subsections will go over the most important parts of these scripting components that are vital for the game to work as intended. Most of the methods and behaviors required/called during gameplay are not detailed in this section, however, and are instead given their own sections later in this chapter.

5.1.1 Game Controller

The *Game Controller* class is the most complicated class in the game at present, with the second being the *Object Controller*. This class is responsible for the following:

- Setting the action cycle state and knowing when to transition between action cycle states (with the exception of *manipulation* to *connection*, section 5.5)
- Instantiating puzzle pieces in the scene from either an *instantiation button* (section 5.4) or a *design rule* (section 5.8)
- Saving *design rules* to the *design schema* (section 5.8)

- Player undo (section 5.9)
- All actions done in both the *instantiation* (section 5.4) section and *exploration* (section 5.11) action cycle states.

The *Game Controller* class has seen the most change throughout development as just about every method contained within the other scripting components for the game controller game object started off as part of the *Game Controller* class before it was decided that they should either get their own class or be a part of a different class

5.1.2 Instantiate Dictionary

The *instantiation dictionary* is used to help link puzzle piece prefabs and game object instances to their corresponding *instantiation button* by the *prefabID* saved to the puzzle piece's *Object Controller* script (section 5.2.1 Object Controller Script). This dictionary is used whenever an instance of a puzzle piece game object in the scene is destroyed and the *instantiation UI panel* needs to be updated to reflect that change (like undo described in section 5.9). This dictionary is also vital for when the game needs to instantiate a puzzle piece game object outside of the *instantiation* action cycle state. For this type of instantiation, the *Game Controller* script uses a *prefabID* from either a *design rule* or a list from the data loaded from a pre-existing save, to figure out which puzzle piece prefab to instantiate. The figure below is a graphical representation of the *instantiate dictionary* used in the Martin House puzzle level.

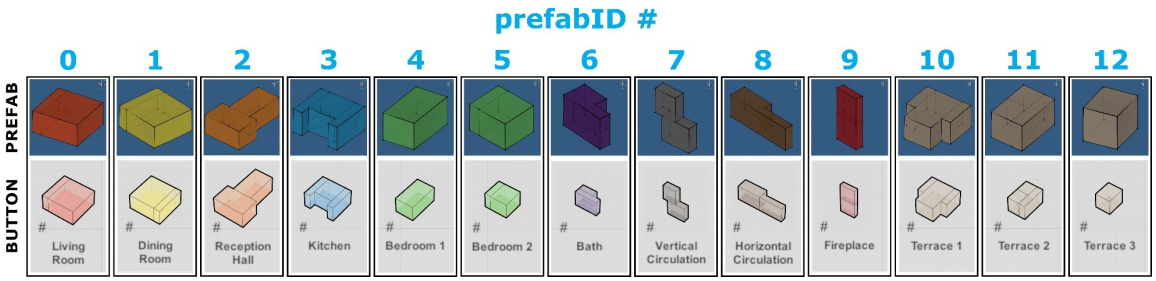


Figure 5-2 Instantiate Dictionary for Martin House Puzzle

Source: Author

5.1.3 In-Game Menu

The in-game menu is opened using a button located at the tope left of the game window/screen. Clicking on that button both makes the in-game menu visible

and interactable to the player and blocks raycasts, preventing the player form being able to interact with anything else in the scene.

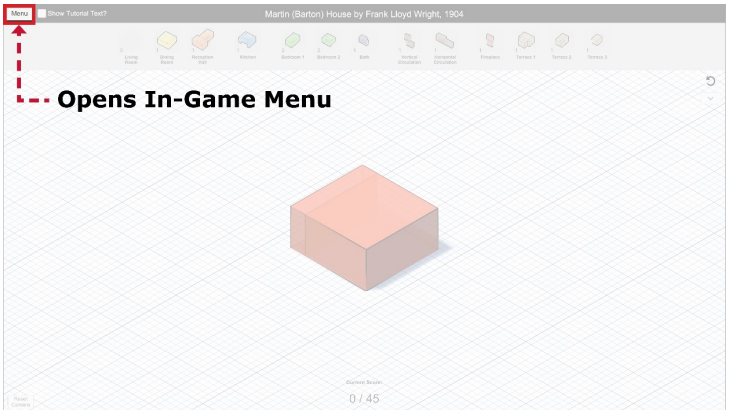


Figure 5-3 In-Game Menu Button

Source: Author

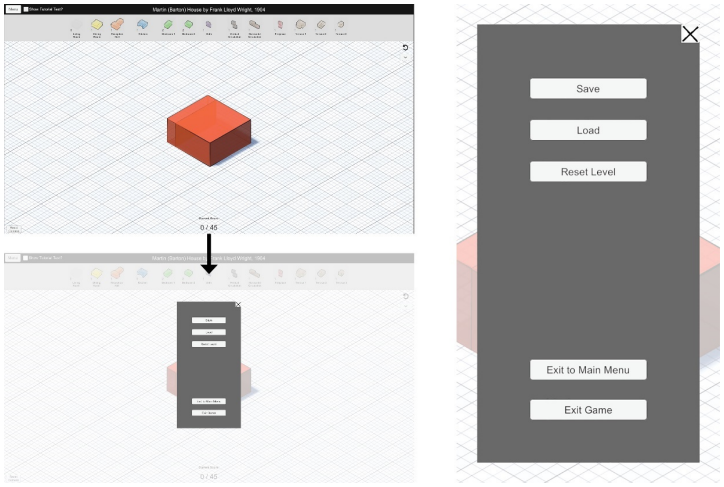


Figure 5-4 In-Game Menu

Source: Author

The in-game menu that the player can access within the puzzle piece scene is very much an early prototype and many elements of it are placeholders. It does, however, currently allow a player to save, load, reset, and exit their game. I will not go over how these methods currently work in this document as they are all still very rough and not critical to current gameplay functionality, but they do work.

5.2 Puzzle Piece Game Objects

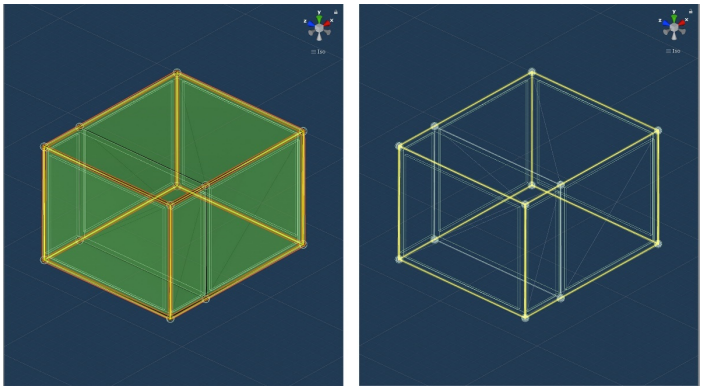


Figure 5-5 Puzzle Piece Prefab Example (Bedroom 2)
Source: Author

Each puzzle piece game object is an empty game object assigned components which handle actions and behaviors done to or by the puzzle piece. Each puzzle piece game object in a level is constructed by assigning a number of components and child objects to an empty game object, which is then saved as a prefab within the game files for the player to instantiate during runtime. In this section I will detail the general process I went through to generate the puzzle pieces for the Martin House level, starting just after the last abstracted iteration was developed.

Each puzzle piece game object is assigned the following components:

- *Transform.* Handles information about the game object's position, rotation, and scale[1]
- *Line Renderer*. Takes an array of points located in 3D space to draw a continuous line and is used here to outline each puzzle piece's overall shape[2]
- *Object Controller Script*. A custom class that holds information about each puzzle piece and handles all puzzle piece behaviors not addressed by the other components.
- *Object Mover Script*. A custom class that is used to manipulate the *transform* component.

There are three types of child game objects attached to each empty puzzle piece game object. They are:

- *Shape Child Game Objects*
- *SnapPoint Game Objects*
- *ConnectSrf Child Game Objects*

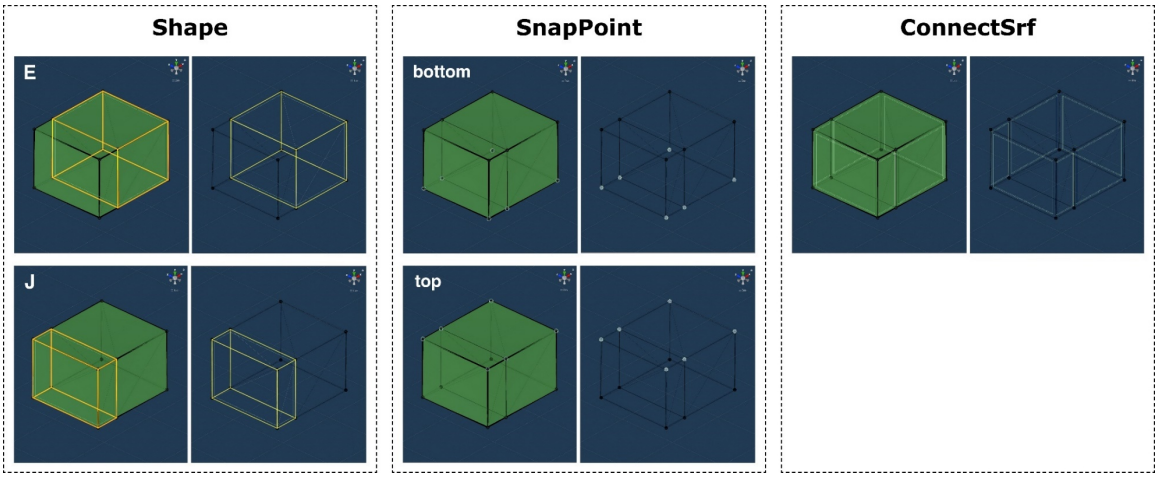


Figure 5-6 Children of Puzzle Piece Prefab Example (Bedroom2)

Source: Author

Subsections 5.2.1 and 5.2.2 will go over the two custom classes assigned to each puzzle piece game object. Subsections 5.2.3 - 5.2.5 will go into detail about the purpose for each of the child game object types and how they are constructed within the *Unity* editor.

5.2.1 Object Controller Script

The overall purpose of the object controller script is to handle information specific to each puzzle piece instance. This class holds the following public variables:

- *Floor Level*. An integer used to specify the floor the puzzle piece is located.
- *Room Height*. A float that specifies how tall a puzzle piece is
- *Correct position*. A bool that looks if the player placed the puzzle piece in the correct position. Defaulted to be false.
- *Correct Rotation*. A bool that looks if the player placed the puzzle piece at the correct rotation. Defaulted to be false.
- *Is Symmetrical*. A bool defined in the editor to signify if a puzzle piece is symmetrical, that mirroring the puzzle piece across the xy and yz plane does not alter its appearance.
- *Connection Possible*. A bool used for the *connection* action cycle state and set up in the *manipulation* action cycle state.

- *Puzzle Piece Info*. A custom class containing information about the puzzle piece that is used when comparing two or more puzzle pieces as well as a means of identifying the puzzle piece in other classes and other instances of the *Object Controller* class.
- *Puzzle Piece State*. A generic enumeration used to designate the state of a puzzle piece (section 5.3.2)
- Materials used to adjust the visual appearance of each puzzle piece to reflect the *Action Cycle State* and the *Puzzle Piece State*.
- Arrays for each of the child object types
- *Instantiation Button*. A reference to which button was used to instantiate a particular puzzle piece
- *Connected Puzzle Pieces.* A list of which puzzle pieces a particular puzzle pieces has been connected to during the *connection* action cycle state.

The *Info* class contains defines the following variables:

- *Precedent.* A generic enumeration that is used to designate the specific precedent a game object belongs to.
- *Function.* A generic enumeration that assigns a specific function to a puzzle piece (like "kitchen" or "bedroom").
- *Shape ID.* A string containing information about the individual shapes that make up the puzzle piece.
- *Prefab ID.* An integer used to identify the prefab for each puzzle piece.
- *Instance ID*. An integer used to identify a specific instance of each puzzle piece, also identifies when it was instantiated within the scene.

The *Object Controller* class itself is used to preform all actions done to a puzzle piece by a player both within and outside of the core action cycle with the exception of the *manipulation* action cycle state which is handled in the *Object Mover Script*. This class is, however, used to enable the *Object Mover Script* assigned to a

puzzle piece game object upon entering the *manipulation* action cycle state, which is disabled by default.

5.2.2 Object Mover Script

Object Mover is a custom class that handles changes made to a puzzle piece during the *manipulation* state of its action cycle, with the exception of corner-to-corner snapping which is handled in the *Snapping* class (section 5.5.2 Corner-to-Corner Snapping) assigned to each *Snapping Child Game Object*. The *Object Mover* class is also responsible for moving from the *manipulation* state to another action cycle state (section 5.5).

5.2.3 Shape Child Game Objects

The *Shape Child Game Objects* hold the 3D models for each of the *basic* or *unique* shapes for each puzzle piece. These child game objects are also "empty game objects" similar to the puzzle piece game objects and are saved as prefabs within the game's asset folder. Each *shape child game object* is assigned the tag "Shape" so that the game will recognize the game object as a *shape child game object.*

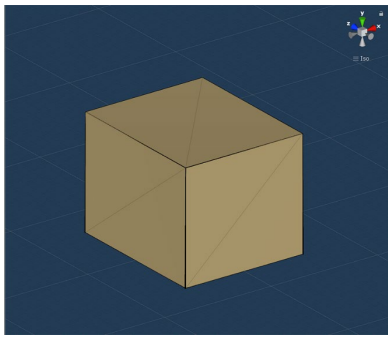


Figure 5-7 Shape Prefab Example
Source: Author

A *Shape Child Game Object* has the following components:

- *Box Collider*. Not currently used for anything, but intended for overlap detection outlined in section 6.6.3
- *Line Renderer.* The same as the line renderer component in the puzzle piece game object but is visually thinner and used to graphicly distinguish the individual shapes comprising each puzzle piece.
- *Shape Controller Script*. A custom class that holds information about its dimensions and an ID.

In chapter 3, Rhino was used to create a 3D model of a puzzle where each 3D geometry represented a *basic* or *unique* shape belonging to each puzzle piece either on its own or as part of a *compound* or *unique compound shape*. One of each shape

size was then exported from *Unity* as a .dea file and saved in the assets folder in the *Unity* project. This allows the files to be read in the *Unity Editor.* These .dea files are then made children for a corresponding *shape child game object*.

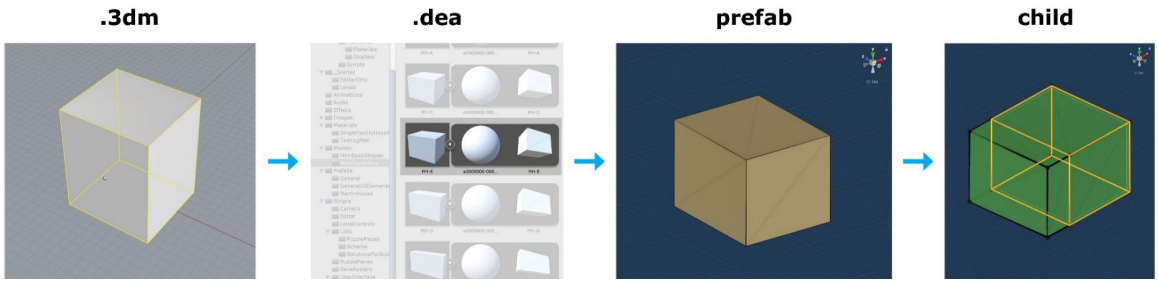


Figure 5-8 Rhino to Child of Puzzle Piece Game Object Process (Shape E)

Source: Author

During the process of abstraction, 3D massing models for the precedent being abstracted are created in Rhino. This was done primarily to initial feasibility testing for each specific abstracted iteration to see what constraints within the abstraction process made solving the precedent puzzle in addition to generating alternative designs easier or harder. This is not the only purpose for using Rhino as the individual 3D shapes created within Rhino are used to create the models used in Unity. Each individual 3D shape within the abstraction Rhino model exported as a COLLADA file (.dea), making sure that the origin point within the Rhino model space was located exactly at the bottom center of each shape. These COLLADA files where then imported into the Unity project where they serve as both the mesh used for rendering and the mesh collider used in calculating the game object's physics. These two different types of meshes are nested within each other, with game object containing the actual model (the mesh being rendered) as a child of an empty game object which contains the physics components (the mesh collider). The empty parent game object serves as the basic shape model later form the puzzle piece game objects.

5.2.4 SnapPoint Child Game Objects

SnapPoint child object are used for corner-to-corner snapping in the *manipulation* action-cycle state. Each of these *SnapPoint child game objects* has one child game object, a tiny 3D sphere model which can be turned on or off to help make the object corners more visually distinct, and has the following three components:

- *Transform.* Handles information about the game object's position, rotation, and scale[3]
- *Sphere Collider.* Used to trigger corner-to-corner snapping between two *snapping child game objects*.
- *Snapping Script*. A custom class used for corner-to-corner snapping detailed in section 5.5.2

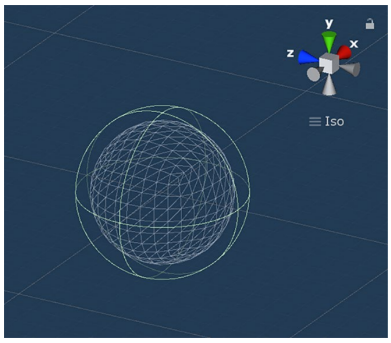


Figure 5-9 Snapping Prefab

Source: Author

While all *SnapPoint child game objects* have a *Snapping script* component assigned to it, the *Object Mover* assigned to the parent puzzle piece game object either enable or disables this component depending on the location of the *SnapPoing child game object* within the puzzle piece as a whole, the action cycle state, and the puzzle piece state. Each *snapPoint child game object* is tagged "SnapPoint" so that they can recognize one another for corner-to-corner snapping

Currently only the *SnapPoint child game objects* located at the bottom corners of the *shape child game objects* have their *Snapping script* component enabled when the puzzle piece is being manipulated in the *manipulation* action cycle state.

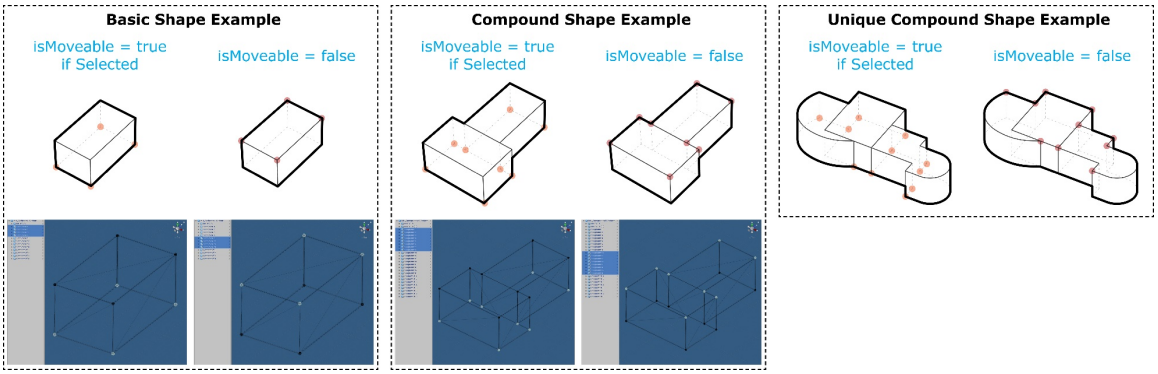


Figure 5-10 SnapPoint Location Relating to Movability Examples

Source: Author

5.2.5 ConnectSrf Child Game Objects

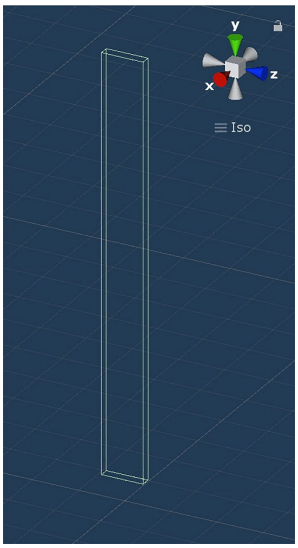


Figure 5-11 ConnectSrf Prefab
Source: Author

These child objects are used to detect if one puzzle piece game object is up against another, i.e. that they have side surfaces that are touching each other. Each *ConnectSrf child game object* has the following components:

- *Transform.* Handles information about the game object's position, rotation, and scale[4]
- *Box Collider*. A trigger collider used to detect if a *placed* puzzle piece is adjacent to a *selected* one.

The *ConnectSrf* prefab is tagged "ConnectSrf" for the same reason as the *SnapPoint* has its tag. A *ConnectSrf child game object* is place at the outer edges of each *shape child game object* and scaled to match its length. The *ConnectSrf* prefab is set up so that instances of its *transform.rotation* and *transform.scale* can be adjusted so that it will run along most of a *shape child's* length. This is done so that the *box colliders* will not trigger if only the edges of two puzzle pieces are touching.

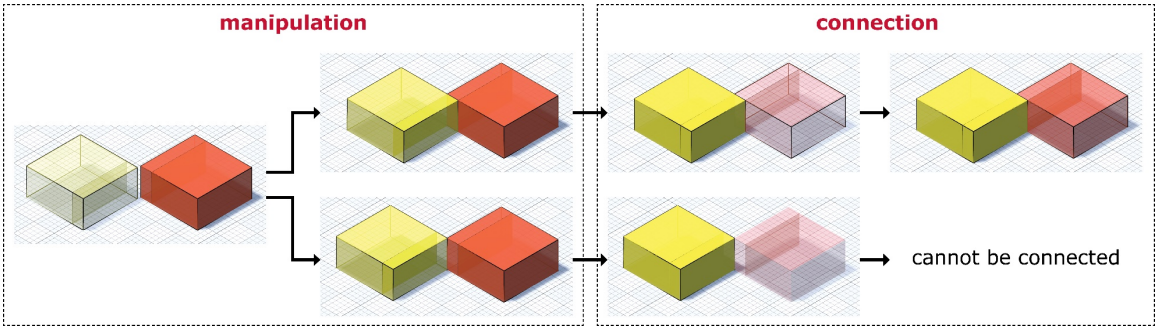


Figure 5-12 Connections Require Overlapping Surfaces
Source: Author

5.3 Action Cycle Overview

The player action cycle is used to define the actions a player does in the game and the order in which they are performed. The key for all flowchart diagrams can be found in Appendix A. Each of the steps in red text define an action cycle state which

is set up in the *GameController* script component of the *GameController* empty game object. There are currently five action cycle states:

- *Instantiation.* State where player selects a puzzle piece to instantiated within the scene.
- *Manipulation.* State where a player places the newly instantiated puzzle piece.
- *Connection.* State where a player designates adjacent puzzle pieces as connected or not
- *Exploration.* State where a player explores past choices/actions using the generated *design schema.*
- *Other.* Currently used when instantiating puzzle pieces from a previous save, a miscellaneous state for any feature that cannot work within the other four states.

Within the different action cycle states are puzzle piece state which specify what actions are being done to them and how they should appear graphically to the player. The five puzzle piece states:

- *Selected.* Puzzle piece belonging to current action cycle
- *Placed.* Puzzle piece that has already had its action cycle
- *Connected.* A *placed* puzzle piece that is going to be connected to the *selected.*
- *Add.* Puzzle piece being added to scene from *exploration* of schema
- *Subtract.* A *placed* puzzle piece to be destroyed/removed from the scene.

The following subsections will go over the action cycle and puzzle piece states in more detail.

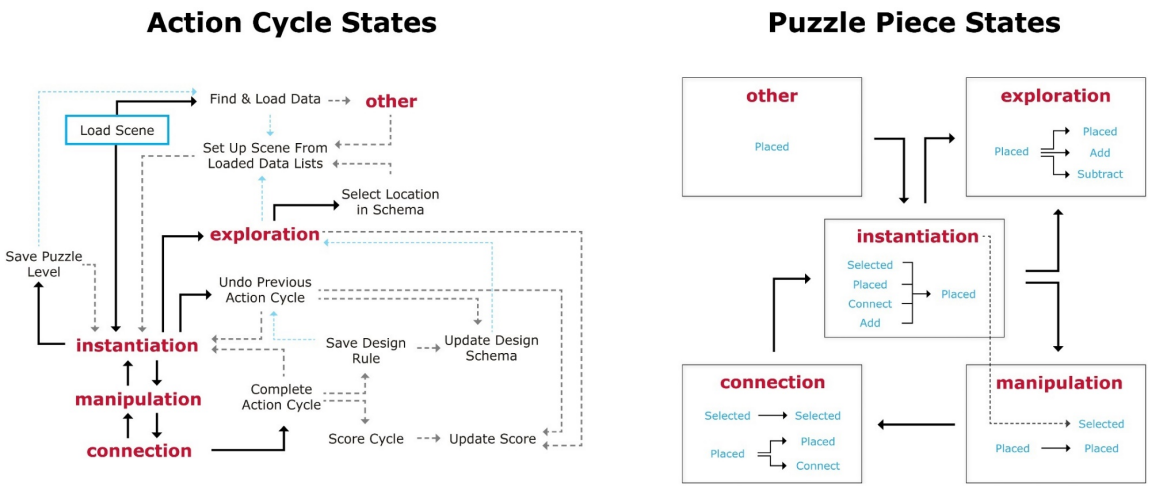


Figure 5-13 Action Cycle & Puzzle Piece States Overview

Source: Author

5.3.1 Action-Cycle States

As the player acts within the level, the game switches between various states depending on where in the player action cycle they are. Within each action-cycle there exists a *selected* puzzle piece which the action-cycle belongs to. It is this *selected* puzzle piece that the player is acting upon. The states built into the action-cycle are explained in detail later in this section, but in summary there are two main types of Action-Cycle states: core states and auxiliary states. The core action-cycle states are:

- Instantiation
- Manipulation
- Connection

Instantiation refers to the state where the player chooses a puzzle piece to instantiate within the current scene. This puzzle piece game object then goes on to be the selected game object belonging to the action-cycle. This state acts as

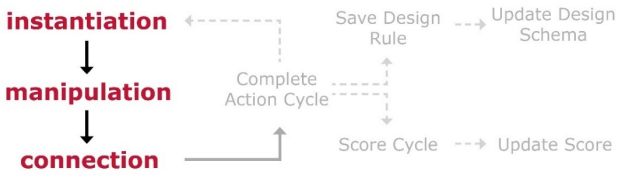


Figure 5-14 Core Action Cycle States

Source: Author

both the entrance and exit point for the player within the action-cycle.

Manipulation refers to the state in which the player picks the position and rotation for the action-cycle's selected puzzle piece.

Connection refers to the state where the player designates other puzzle pieces within the scene and adjacent to the action-cycles selected puzzle piece that the player wished to designate as being "connected" to the selected.

The auxiliary action-cycle states refer to the states that are needed to preform actions outside of the core player actions of picking a puzzle piece, moving it into position, and defining how it relates to the rest of the puzzle pieces within the scene. Currently there are only two auxiliary states, and they are:

- Exploration
- Other

Exploration refers to the state where the player can explore the design schema generated by their actions during gameplay/

Other currently denotes an action-cycle state where the game is loading and is used as a default for when the game specifically needs to ignore player input.

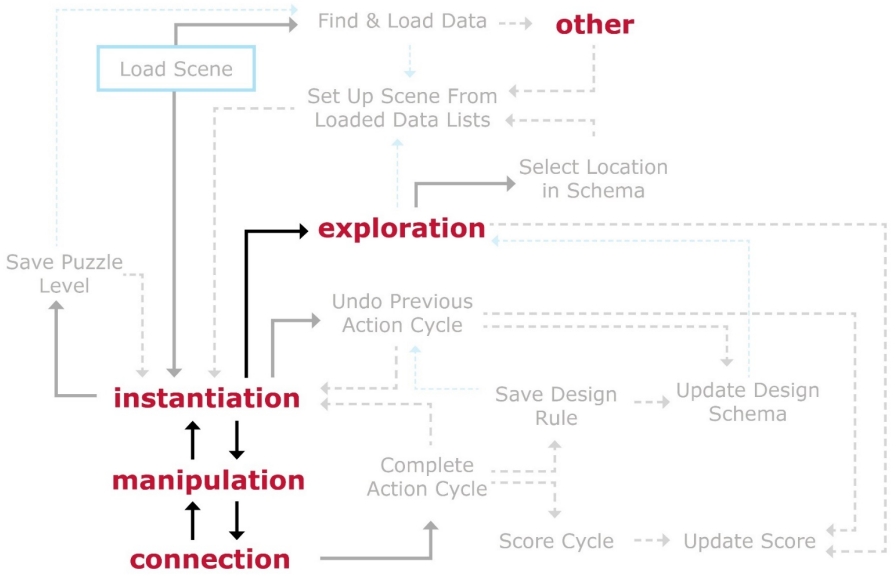


Figure 5-15 Action Cycle States

Source: Author

5.3.2 Puzzle Piece States

Just as the game uses action-cycle states to help differentiate between the actions available to the player at a given point within the action-cycle, the puzzle pieces also have specific states that they move between. These primarily serve as a means to help adjust materials to convey information about the puzzle pieces to the player and as a means of specifying the types of actions the player can perform with each puzzle piece. These states can also be sorted into two different groupings: core states and exploration states.

The core puzzle piece states are the primary ones used within the game as a whole and are the only states used within the core action-cycle states. The core puzzle piece states are:

- Placed
- Selected
- Connected

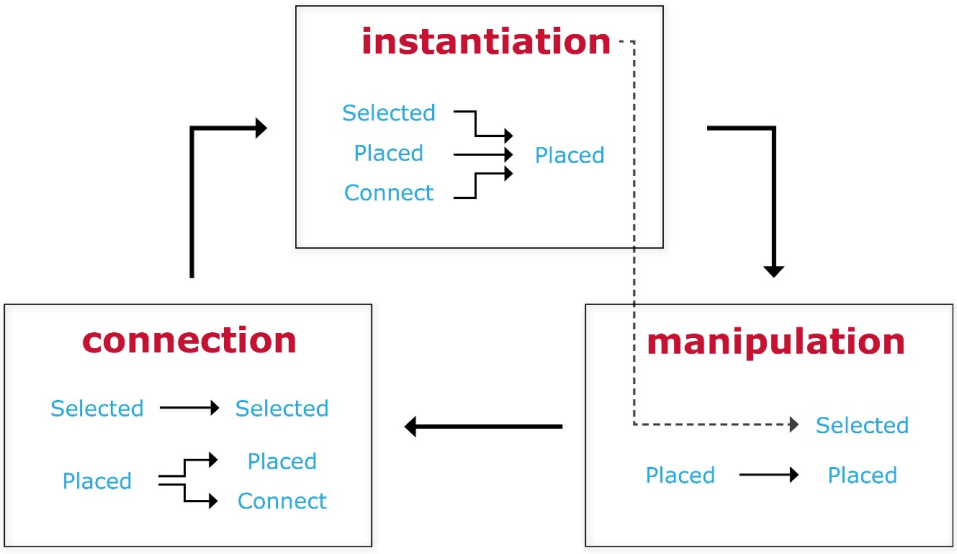


Figure 5-16 Overview of Core Puzzle Piece States

Source: Author

Placed is the default state a puzzle piece has and is used to denote puzzle pieces that have already gone through their action-cycle.

Selected is the state used to denote the current action-cycle's puzzle piece, i.e. the one the player is currently acting upon.

Connected is a special case state only used within the *connection* action-cycle state and is used to denote which puzzle pieces are "connected" to the current *selected* puzzle piece. "Connected" refers to the player defining that a person moving through the design could move directly from one puzzle piece "room" to another.

The exploration puzzle piece states are special case states that only are used within the *exploration* action-cycle state. They are used in addition to the core puzzle piece states. There are only two exploration puzzle piece states and they are:

- Add
- Subtract

Add is the state used to denote puzzle pieces which would be added to the scene based off the design schema.

Subtract is the state used to denote puzzle pieces which would be deleted from the scene during special cases of the *exploration* action-cycle.

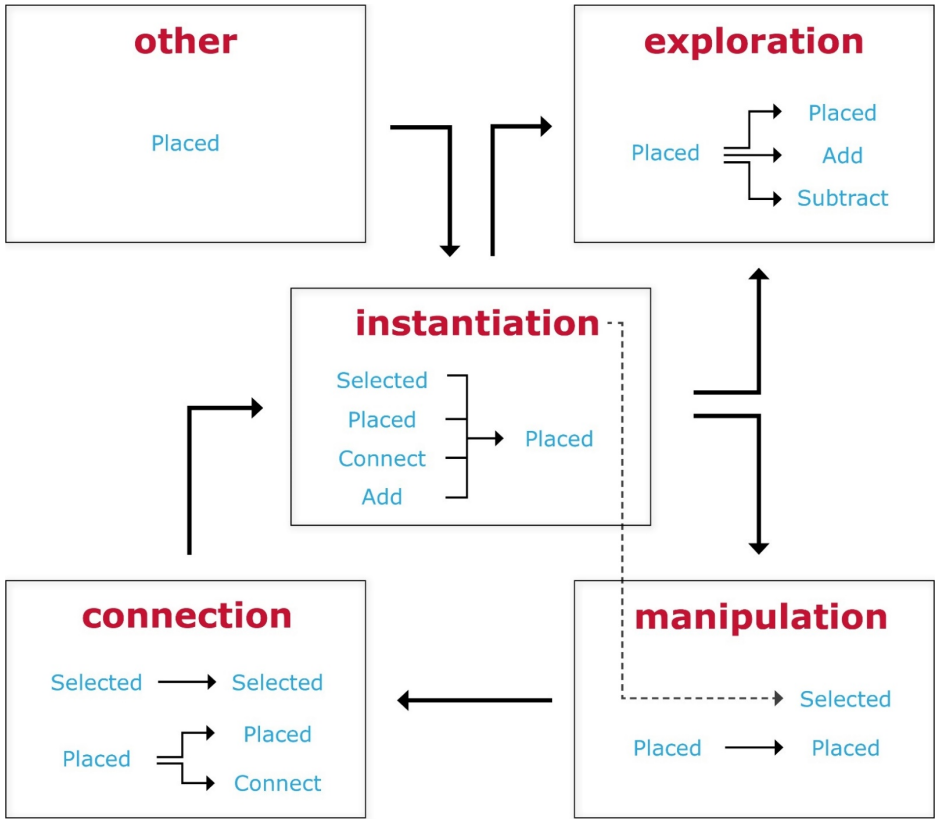


Figure 5-17 Overview of All Puzzle Piece States

Source: Author

As can be seen in the figure above, puzzle pieces states are linked to the current action cycle state. Puzzle pieces automatically are given the *selected* puzzle piece state upon being instantiated within a scene, except in the case where there are no other puzzle pieces in the scene or if the action cycle state is set to either *other* or *exploration.* There are two types of cases where a puzzle piece is told to change its state. The first case is when entering a specific action cycle state outside of instantiating a puzzle piece. This occurs when entering the *instantiation* and *manipulation* action cycle state. In the *instantiation* state any puzzle piece whose state is not currently *placed* is set to *placed.* Changes to puzzle piece states upon entering the *manipulation* state only occur if the player comes from the *connection* state where any puzzle piece whose state is *connected* is set to *placed.*

The second case where a puzzle piece's is when a player preforms certain actions to a puzzle piece within a given action cycle state. In the *connection* state a puzzle piece can toggle between *placed* and *connect*. In the *exploration* state a puzzle piece can toggle between *placed* and *subtract* as well as between *add* and *subtract.*

Figure 5-18 Puzzle Piece Material Adjustments Example

Source: Author

Whenever a either a puzzle piece's state or the action cycle state is changed, a method is called to assign a specified material and enable or disable the line renderer components in both the parent and children based on the action cycle state and the puzzle piece state.

5.4 Instantiation

Instantiation is the action cycle state where a player selects a puzzle piece to instantiate within the scene. Game objects are instantiated within a loaded scene during runtime using the Instantiate( ) method native to *Unity*. This method takes a few different parameters, the version used for this project takes a game object, a Vector3, and a Quaternion. The instantiate method works by creating a clone of the given game object whose position equals the given Vector3 and rotation equals the

given Quaternion. The current build of the game uses this method to place prefabs of the puzzle pieces in a scene during runtime.[5]

While the instantiate method has several uses within the game at this time, this sections is specifically looking at its use in the player instantiation of puzzle pieces from a list set up in the editor during development. Information about the puzzle pieces (their form, function, and number remaining) is conveyed to the player through the UI. Specifically UI images and text components assigned to button game objects located at the top of the game screen. Each button game object is also assigned an instance of the *ButtonController* class which tells each button how many puzzle pieces it can create in total.

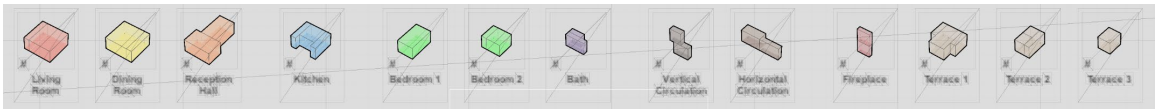


Figure 5-19 Screenshot of Instantiation Button Panel for Martin House in Unity Editor
Source: Author

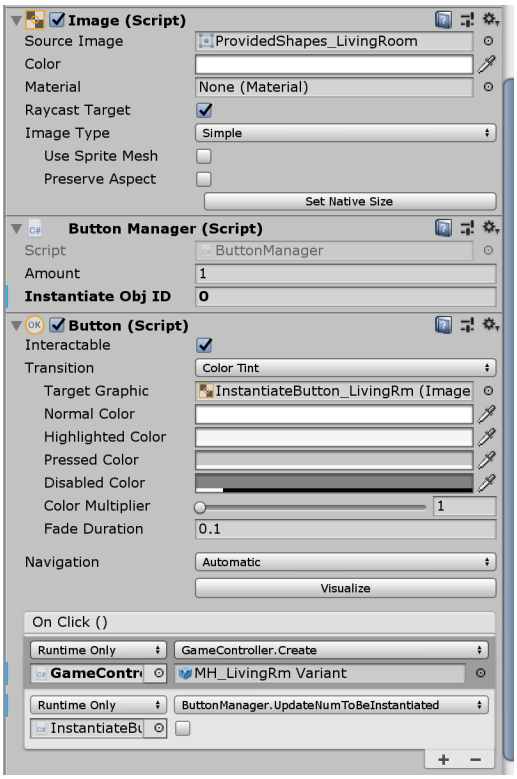


Figure 5-20 Instantiation Button Components
Source: Author

The play does this by clicking on a button in the *instantiation UI panel* that corresponds to the puzzle piece they wish to instantiate. Each instantiation button game object has the following key components:

- *Image.* The image the button displays
- *Button Manager Script.* Custom class that tells each button how many puzzle pieces it can instantiate and the *prefabID* its corresponding puzzle piece
- *Button Script*. Calls a selected method upon being clicked. The component that allows it to be a button.

A button game object has a button component assigned to it that detects when it is clicked on by the player. When this happens, it calls one or more public methods set up in the inspector window in the editor.

Buttons can be toggled between being interactable and not, and different visuals can be set to help the player distinguish if a button is interactable or not. Each instantiation button game objects

The button game objects in the instantiation UI panel are set up so that when a player clicks on it, a method is called in the GameController class that handles setting up the parameters taken by the Instaniate( ) method before calling that method to instantiate a corresponding puzzle piece in the scene. If there are no puzzle pieces instantiated within the current puzzle scene, like if the player had just loaded the scene for the first time, the game restricts the puzzle pieces available to the player by setting every button whose assigned puzzle piece is not located on the first floor set to being not interactable. This is done because the first puzzle piece game object instantiated within the scene becomes the *player initial object* and is placed on the ground at the origin.

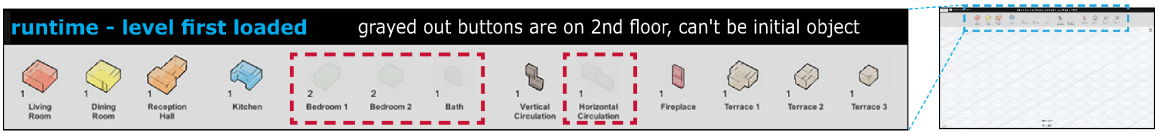


Figure 5-21 Instantiation UI when Selecting Player Initial Object

Source: Author

If there is one or more puzzle pieces in the scene any button that corresponds to a puzzle piece that has yet to be places is interactive. When the player clicks on any interactive button at this time the game will instantiate the corresponding puzzle piece in the scene, basing its instantiation position and rotation based off combination of the player's mouse position and predefined variables set up in the editor. The game uses the mouse position to determine the x and y position values (i.e. its horizontal position) with the y position value (i.e. the vertical position) and rotation set up in the editor. The y position value is automatically calculated using the floor the puzzle piece is assigned to and the height of the floors using the following equation:

$$transform.position.y = (assigned\ floor - 1) \times floor\ height$$

When a puzzle piece is instantiated in this manor the game designates it the *selected object* for this action cycle meaning that all actions done within this cycle are acting on or in direct relation to that puzzle piece. The game will then automatically move

to the *manipulation* action cycle state, graying out an disabling the instantiation buttons.

As the player instantiates puzzle pieces, the game tracks how many of each puzzle piece is left, displaying the number remaining at the bottom right corner of each button. When all of a specific puzzle piece have been instantiated the button is disabled so that the player does not instantiated any unnecessary duplicates, letting the game inform the player of how close they are to completing the current puzzle. The game also registers whenever a puzzle piece is removed from the scene (i.e. is destroyed/deleted) and updates the instantiation UI accordingly, re-enabling the puzzle piece's corresponding button if it had been disabled.

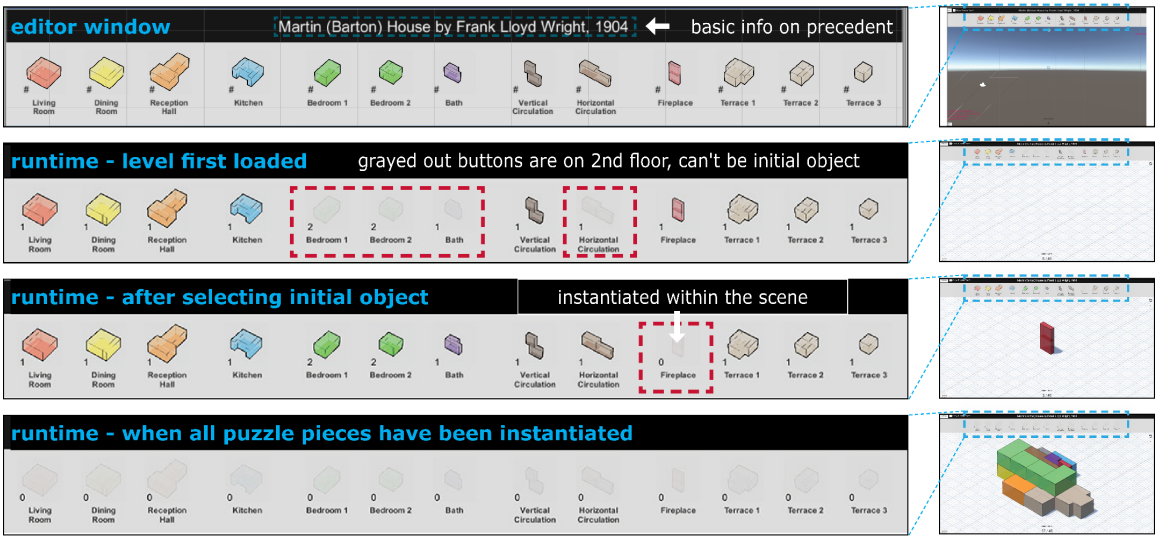


Figure 5-22 Instantiation Button Interactions

Source: Author

If there are no other puzzle piece game objects current in the scene, the next puzzle piece the choose to instantiate is designated the *initial player object* and the core action cycle skips the *manipulation* and *connection* states, completing a core action cycle before starting a new action cycle at the *instantiation* action cycle state. If there are one or more puzzle pieces in the scene, then the puzzle piece selected by the player for instantiation is designated the

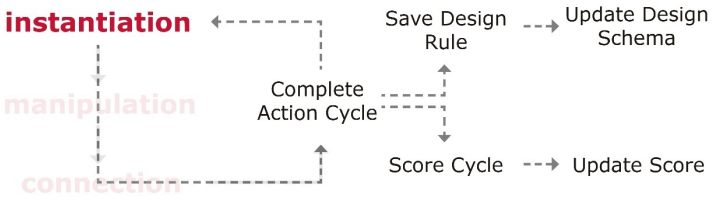


Figure 5-23 Core Action Cycle for Player Initial Object

Source: Author

selected puzzle piece and the game progresses to the *manipulation* action cycle state.

5.5 Manipulation

Manipulation is the action cycle state where a player determines the *transform.position* and *transform.rotation* for the action cycle's *selected* puzzle piece game object. A puzzle piece's puzzle piece state is set to *selected* when it is first instantiated within a scene within the core action cycle, provided that there is at least one other puzzle piece in the scene at that time. *Manipulation* is currently done through behaviors found in three custom classes:

- *Object Controller.* Script component of each puzzle piece game object
- *Object Mover.* Script component of each puzzle piece game object
- *Snapping.* Script component of *SnapPoint* child game object

The *Object Controller* class is used to determine which puzzle pieces is being manipulated by looking at its puzzle piece state. If a puzzle piece's state is set to *selected* upon entering the *manipulation* action cycle state, then the *Object Controller* enables the *Object Mover Script* component for that puzzle piece.

The *Object Mover* class moves the and rotates the *selected* puzzle piece based on the player's mouse position and input keys. It also checks to see if it is snapped to another puzzle piece game object in the scene using the *isSnapped* bool (which is defaulted to equal false) to determine if the game can progress to the *connection* action cycle state. Sub-section 5.5.1 outlines translation and rotation of the *selected* puzzle piece.

Subsection 5.5.2 and 5.5.3 outline the constraints on translation and rotation in the current build of the game, including the *Snapping* class is responsible for corner-to-corner snapping.

5.5.1 Translation & Rotation

Upon entering the *manipulation* action cycle state, if a puzzle piece's puzzle piece state is set to *selected*, then its *Object Mover* script is enabled. This allows the

player to move and rotate said puzzle piece game object around in the scene. Position is controlled by the player's mouse position and rotation is controlled by the ',' and '.' keys, which is common in games have some kind of player "build" mode like the Sims. The puzzle piece's *transform.position* is adjusted in the *FixedUpdate*( ) method in the *Object Mover.* The method a Raycast to figure out what portion of the ground plane in the scene the mouse is over. That information to set the *x* and *z* values for the puzzle piece's *transform.positon* (*y* value is currently set by the Object Controller script where:

$$transform.position.y = (assigned\ floor\text{-}1) \times floor\ height$$

The *FixedUpdate( )* method in the *Object Mover* is also used when checking for player inputs used to rotate the *selected* puzzle piece game object and to exit the *manipulation* action cycle state. The player can use either the 'enter' key or left mouse button to exit the *manipulation* action cycle state and progress onto the *connection* action cycle state. They can also exit out of the *manipulation* state back to the *instantiation* state by using the 'escape' key.

Constraints currently exist within the game which tell the game controller when it is possible for the player to progress onto the *connection* action-cycle state and tailor how the puzzle piece's position and rotation are adjusted.

5.5.2 Corner-to-Corner Snapping

Currently manipulation constraint that affect how the selected puzzle piece's position is corner-to-corner snapping. Corner-to-corner snapping is exactly what it sounds like. The player can adjust a selected shape's location by aligning one of its corners with another corner of another shape within the scene. Corner-to-Corner snapping is done in the *Snapping* class which is attached as a script component to the *SnapPoint child game objects* in each puzzle piece game object. Corner-to-corner snapping uses the following variables found in the *Snapping* class:

- *isMoveable.* A bool that tells the *SnapPoint child* if it can move the parent puzzle piece.
- *unSnapMouseDistance*. A float that determines how far away a player's mouse position should get before the puzzle piece unsnaps from a snapped corner.

- *snapTarget*. The *transform* of the *SnapPoint* to being snapped to.
- *snapMousePos*. A Vector2 that saves where the mouse position is at the time a puzzle piece has been snapped and used for unsnapping.

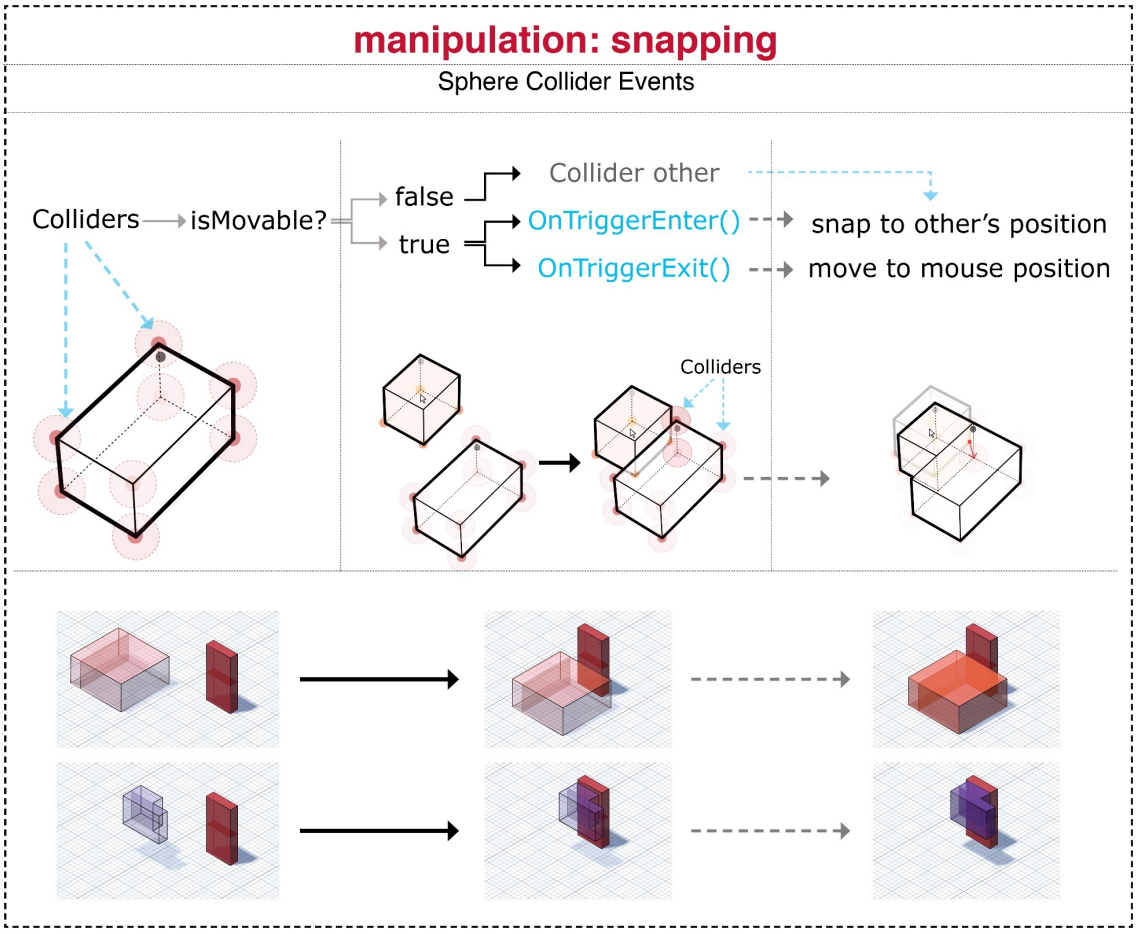


Figure 5-24 Corner-To-Corner Snapping Overview

Source: Author

The *Object Mover* script's *OnEnable( )* method sets the *isMoveable* bool in all the *SnapPoint child game objects' Snapping script* component to true (bool remains set to false for all top *SnapPoint child game objects*, the ones located at the top corners of the puzzle piece) and its *snapTarget* is set to null.

Snapping uses trigger collider events to perform certain actions whenever another collider enters or exits its space. Corner-to-corner is specifically uses the *OnTriggerEnter( )* method which takes some other collider component as a parameter. If the *isMoveable* bool is set to false in the instance of the called

method's class, or if the triggering collider is not tagged as "SnapPoint" the method will simply return, doing nothing. Otherwise *snapTarget* is set to the other colliders transform (which will allow the Snapping script to access the position of the other snapping point) and the *snapMousePos* is set to the current mouse position.

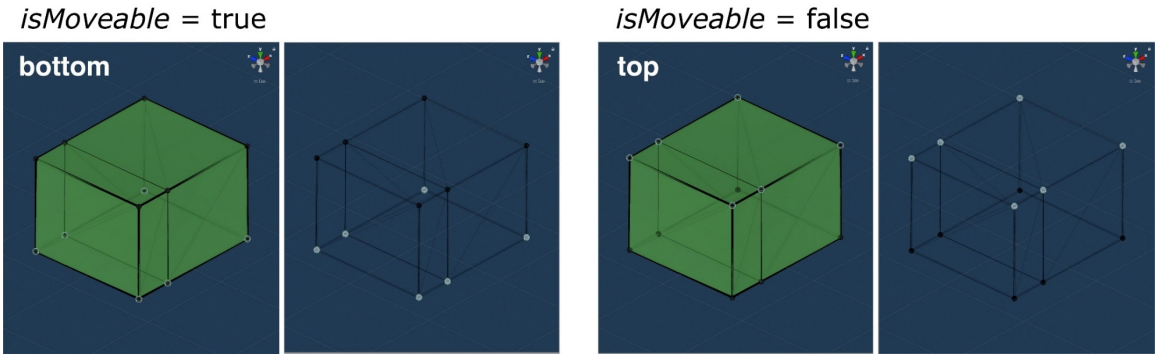


Figure 5-25 SnapPoint Trigger Colliders in Selected Puzzle Piece Example
Source: Author

In the Snapping script's *Update( )* method, if the *snapTarget* variable is not null (i.e. it has been assigned a transform), then the *isSnapped* bool in the *Object Mover* script in the parent game object (which is a puzzle piece game object) is set to true and the parent's *transform* is adjusted using the following equation

transform.parent.position =
snapTarget.position + (transform.parent.position – transform.position)

The game then checks the players mouse position in the *Update( )* method of the *Snapping* class to see if the distance between the *snapMousePos* (defined back in the *OnTriggerEnter( )* method) and the current mouse position is greater than the *unSnapMouseDistanceFloat*. If yes, then the *snapTarget* is set to null once again, the *isSnapped* bool in the parent's *Object Mover* script is set to false, and the *Snapping* script will no longer override the *Object Mover* script when updating the puzzle piece's *transform*.

As the *selected* puzzle piece goes form toggles between being snapped and not, its material updates. This means that when the selected puzzle piece is snapped to another puzzle piece within the scene (i.e. when the *isSnapped* bool in their Object Mover script is set to true) the puzzle piece appears less transparent.

Corner-to-corner snapping does not just alter how a puzzle piece's *transform.position* is manipulated by the player, it also constrains when a player is able to progress to the *connection* action cycle state. The action cycle state can only progress forward to the *connection* action cycle if the *isSnapped* in the selected puzzle piece's Object Controller script is set to true.

Figure 5-26 Manipulation Puzzle Piece Materials by State

Source: Author

5.5.3 Rotation Constraints

Constraints on player manipulation of a puzzle piece's *transform.rotation* is simple. Currently rotation is locked to occurring at 90 degree increments about the y-axis using the *transform.Rotate*( ) method which takes a Vector3 as its parameter.

5.6 Connection

Object connections is the most complicated of the player actions within the core action cycle. There are two steps to object connection: the first happens during the *manipulation* state where the game determines which puzzle pieces are able to be connected to the selected puzzle piece, and the second happens during the *connection* state where the player selects from those puzzle pieces which ones to actually connect to the selected puzzle piece.

The *connection* state requires some initial set up that happens during the prior *manipulation* action cycle state. The following two subsections will go over the set up for the *connection* state and what happens during the *connection* state.

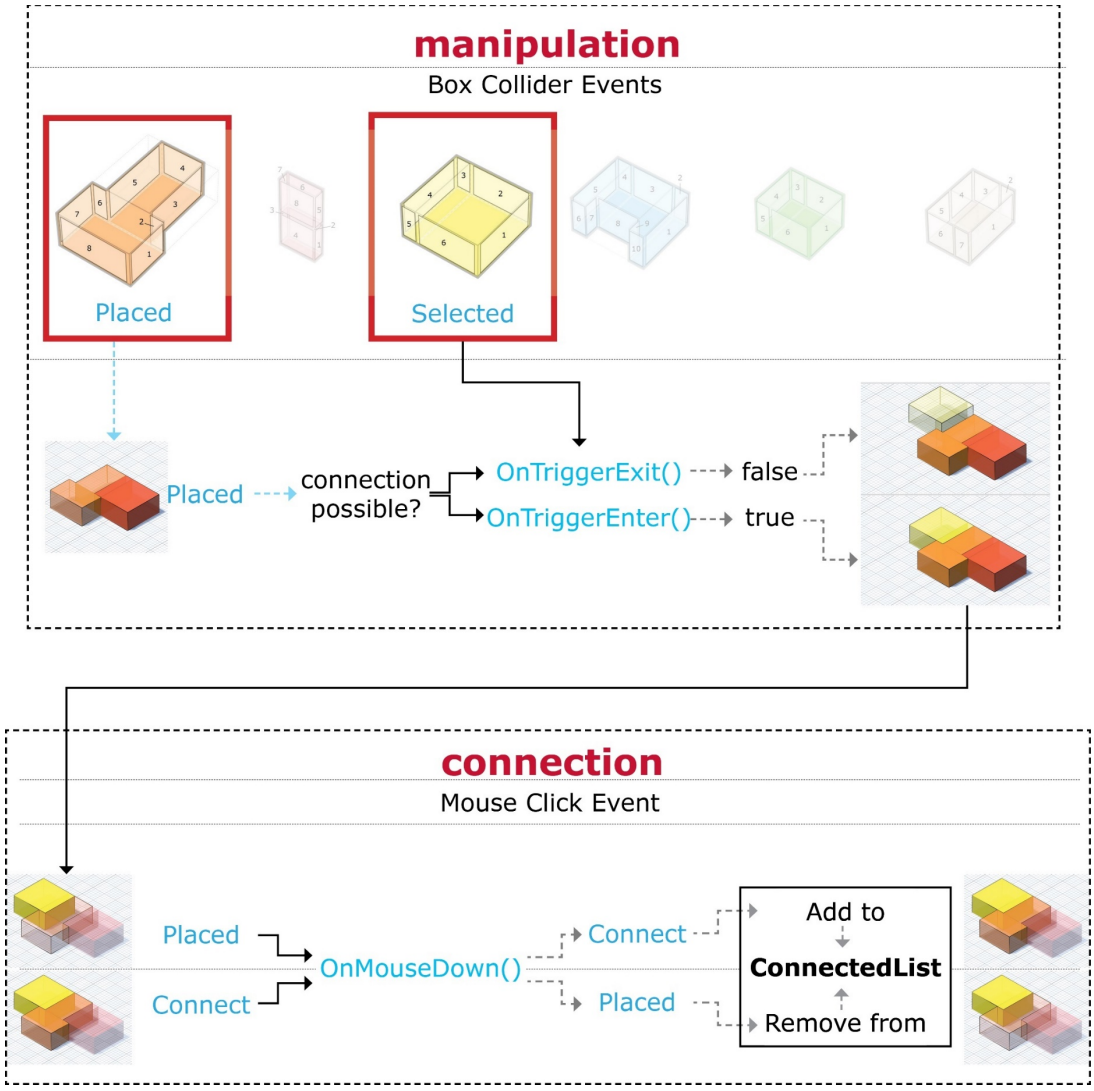


Figure 5-27 Connecting Puzzle Pieces Overview

Source: Author

5.6.1 Connection State Set Up

Set up for the *connection* action cycle state takes place in the *manipulation* action cycle state as the *placed* puzzle pieces in the scene need to detect when they are adjacent to the *selected* puzzle piece (i.e. when one or more of their surfaces are touching). Similar to

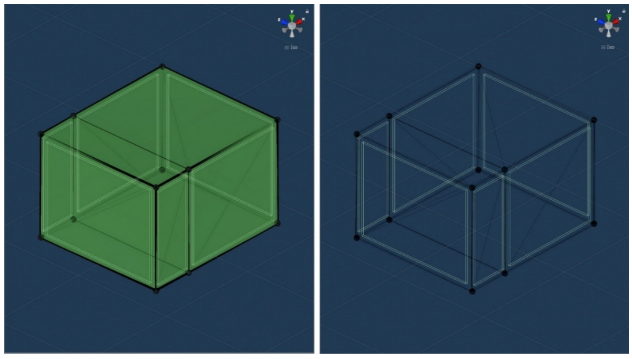


Figure 5-28 Example of Box Colliders for Connection Set Up

Source: Author

corner-to-corner snapping, *connection* set up uses trigger collider. However, instead of using sphere collider components attached to the corners of the *shape child game objects*, *connection* set up uses box collider components attached to the outer surfaces of the *shape child game objects*. These trigger colliders are used to toggle the *connecitonPossible* bool in the parent puzzle piece game object's *Object Controller* script. This information is not communicated visually to the player until they enter the *connection* action cycle state as it is not relevant until that point.

5.6.2 Connection State Actions

The actual connecting is done during the *connection* state. The process of connecting two puzzle pieces looks at the following variables in the *Object Controller* script component for any *placed* puzzle piece game object in the scene:

- *connectionPossible.* The bool that was set up in the *manipulation* action cycle state that lets a puzzle piece know if its adjacent to the *selected* puzzle piece game object
- *Puzzle Piece State*. A generic enumeration that can be toggled between *placed* and *connected* based on player input controls.
- *ConnectedPuzzlePiece List*. A list holding the information about all the puzzle pieces it is connected to and information about scoring.

ConnectedPuzzlePiece is a custom class that is forms the list of *ConnectedPuzzlePieces* found in each instance of the *Object Controller* script and contains the following variables:

- *ConnectedInfo*. A custom class containing information about the puzzle piece being connected to this puzzle piece.
- *hasBeenScored.* A bool that is used to determine if this connection has already been scored so that the same connection does not keep getting scored.
- correctConnection. A bool that asks if the connection defined by this list item matches one of the 'original' functional connections defined in the in the *Scoring* script.

ConnectedInfo is a custom class that is saved as part of the *ConnectedPuzzlePiece List* and is used to define the following variables from another *Object Controller*'s *Info* detailed in subsection 5.2.1 :

- *Function.* A generic enumeration that assigns a specific function to a puzzle piece (like "kitchen" or "bedroom").
- *Prefab ID.* An integer used to identify the prefab for each puzzle piece.
- *Instance ID*. An integer used to identify a specific instance of each puzzle piece, also identifies when it was instantiated within the scene.

Upon entering the connection state, the game controller script will check to see if the bool connectionPossible is set to true or false, this information is conveyed to the player by adjusting its material and either enabling or disabling the line renderer component. If the bool is set to true, then if a player clicks on the puzzle piece game object than the object's game state is set to connect, updating its material. That puzzle piece object controller adds the relevant portions of its *Info* to the selected puzzle piece's object controller's *ConnectedInfo*, and has the selected object controller do the same for the *ConnectedInfo* list in the object controller for the puzzle piece being connected. If a puzzle piece is clicked on and its puzzle piece state is already set to *connected*, its puzzle piece state is set back to *placed* and is removed the list item containing the selected puzzle piece's *function, prefabID,* and *instanceID* from its *ConnectedInfo* list and removes its f*unction, prefabID,* and *instanceID* from the selected puzzle piece's *ConnectedInfo* list. If the bool *connectionPossible* is set to false, the puzzle piece's object controller script is told to ignore all player inputs.

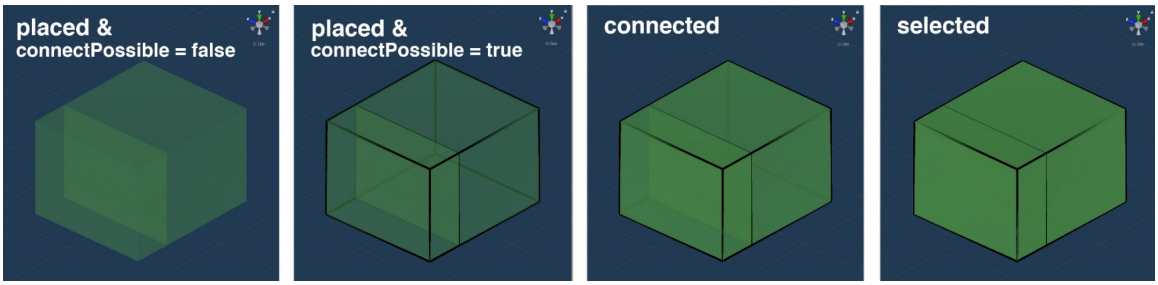


Figure 5-29 Puzzle Piece Connection Status Communicated via Material Transparency & Line Renderer

Source: Author

The player can exit the *connection* state at any time, regardless of whether or not they have specified any connections. They can progress forward by pressing the button located at the top middle of the screen just beneath the *instantiation UI panel* or by pressing the 'enter' key on their keyboard.

5.7 Scoring

Scoring is a way of providing the meaningful feedback necessary for tacit learning.[6] The current scoring system in the game looks at two different groups of criteria:

- *Accuracy of Arrangement*
- *Accuracy of Function Connections*

Scoring can occur at several points during gameplay. The most common instance is when a player has completed a core action cycle as seen below. During these instances the score is added to when upgraded.

Figure 5-30 Scoring Location in Core Action Cycle

Source: Author

Scoring also occurs whenever a player undoes an action cycle or plays in the *exploration* action cycle state. If a puzzle piece is deleted/designated to be deleted, then the score is subtracted from and if a puzzle piece is instantiated into the scene then the score is added to.

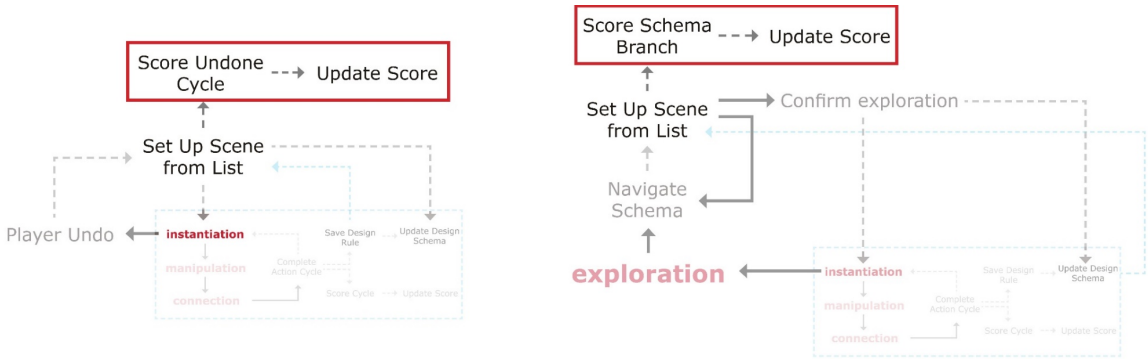


Figure 5-31 Scoring Location Outside Core Action Cycle

Source: Author

All scoring is done using the *Scoring* script component in the *game controller game object*. *Scoring* is a custom class that holds all lists and methods used in scoring in addition to handling any updates to the UI that relate to the player's score (the score text at the bottom center of the screen).

5.7.1 Criteria Lists Overview

There are two scoring criteria, *arrangement* and *function connections*, and each has a corresponding list that the player's score is based off.

- *FormSolutions.* List that holds information about the original's *design arrangement*.
- *FunctionSolutions.* List that holds information about the original's *function connections*.

These criteria lists are looked at in subsections 5.7.3 and 5.7.4 with the specific lists used for the Martin House level located in Appendix C**.**

5.7.2 Scoring Player Initial Object

Scoring a player's initial object (the first puzzle piece that they instantiated within a scene) works a little differently form scoring all the other puzzle pieces and their core action cycles. This is due in large part to how the *FormSolutions* criteria list

was set up. The starting point when designing a Prairie House as defined by Koning and Eizenberg's research is the fireplace. [7] Because of this, when determining the "original solution" for the Martin House to base the scoring criteria off of, the fireplace puzzle piece was the first one instantiated and is therefore the *original initial object*. Therefore, when defining the values for the *FormSolutions* lists the fireplace puzzle piece was placed at the origin with its *transform.position* set to (0, 0, 0,) and its *transform.rotation* set to (0, 0, 0, 0). This means that the values used for each puzzle piece when scoring their position and rotation is equal to their positional and rotational offset from the fireplace.

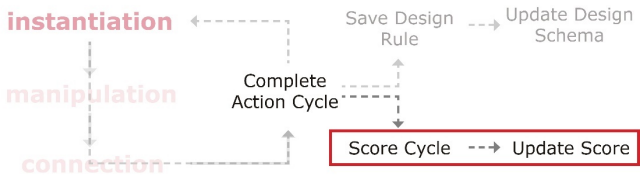


Figure 5-32 Score Player Initial Object in Core Action Cycle

Source: Author

When a *player initial object* is scored the game compares the *Info.funciton* and *Info.shapeID* assigned to the *player initial object's Object Controller* script component to the corresponding values for the *original initial object,* one point is added for each matching variable. The Scoring class then goes to the list item in *FormSolutions* list for the *player initial object* and saves the position and rotation values from the list. These values are used in the calculations done when scoring the arrangement/form of the player's choices. The criteria list marks the criteria in the *FormSolutions* list that corresponds to the *player initial object* as having been met in terms of both its position and rotation.

5.7.3 Design Arrangement

Faithfulness to the original's design arrangement measures how close a player gets to arriving at the same arrangement made during the abstraction process for the precedent (the *original solution*). The more shapes that match the placements found in the *original solution,* the higher the players score. It essentially scores choices the player made in the *manipulation* action cycle state.

Scoring in terms of design arrangement uses the *FormSolutions* list saved to the *Scoring* script component of the *game controller game object*. This list contains the following information about the 'original' solution:

- *Function.* A generic enumeration that assigns a specific function to a puzzle piece (like "kitchen" or "bedroom") taken from the *Info* class. Only used in this list when scoring the *player initial object*.
- *Shape ID.* A string containing information about the individual shapes that make up the puzzle piece.
- *Location.* A custom constructor that takes a Vector3 and a Quaternion as its parameters and translates them into variables not specific to *Unity.* Is used to save a *transform.position* and *transform.rotation*.
- *Is Symmetrical*. A bool defined in the editor to signify if a puzzle piece is symmetrical, that mirroring the puzzle piece across the xy and yz plane does not alter its appearance.
- *positionMet*. A bool that asks if a player has already met the position criteria defined in the list item, defaulted to be set to false.
- *rotationMet*. A bool that asks if a player has already met the rotation criteria defined in the list item, defaulted to be set to false.

When the game scores a puzzle piece's *transform.position* and *transform.rotation* using the *FormSolutions* lists if first finds all indexes in the list whose. When scoring the position, the list of *FormSolutions* being considered is limited to only items where *positionMet* is equal to false and whose *ShapeID* is equal to the puzzle piece's *shapeID.* The game then takes the value located in the criteria list for the *player initial object* and uses it to calculate the position for the puzzle piece game object to be matched with the criteria list. If no matches are found the player does not receive any points added to their score. However, if a match is found from the reduced list, then the player's score goes up by one and that list items *positionMet* bool is set to true. A bool (*correctPosition*) in the *Object Controller* script component of the puzzle piece is also set to true if a match is found in the criteria list.

The process for scoring a puzzle piece's rotation is exactly the same except it uses the *transform.rotation* and if there is a match found, the list item's *rotationMet* bool is set to true and the *correctRotation* bool is set to true in the *Object Controller*

script component of the puzzle piece. One key difference, though, is that if the *IsSymmetrical* bool for the puzzle piece is set to true, it will also check the criteria list for items 180° off.

Whenever a puzzle piece is deleted, this same process works in reverse to deduct points from a player's score if necessary. The *Scoring* class will first check to see if either the *correctPosition* or *correctRotation* bool is set to true, and if so it will check the list items whose corresponding criteria have been met for where the *shapeIDs* match. If a match is found in for either the position or rotation criteria, then one point is deducted from the player's score for each criteria that was matched. The list item's *positionMet* and *rotationMet* bolas are set to false if applicable. This is to prevent the player from being able to keep increasing their score by just placing the same puzzle piece in the scene again and again forever.

5.7.4 Function Connections

Faithfulness to the original's design function connects measures how close a player gets to connecting the same functions that were seen to be connected abstraction process for the precedent (the *original solution*). The more functions connections that match the connections found in the *original solution,* the higher the players score. It essentially scores choices the player made in the *connection* action cycle state.

Scoring in terms of design function connects uses the *FunctionSolutions* list saved to the *Scoring* script component of the *game controller game object*. This list contains the following information about the 'original' solution:

- *Function.* A generic enumeration that assigns a specific function to a puzzle piece (like "kitchen" or "bedroom") taken from the *Info* class.
- *ConnectedFunction.* The same as *Function* but for the function being connected to the *Function* defined above*.*
- *NumRemaining*. An integer specifying how many of this connection type have yet to be made in the scene by the player.
- *TotalConnections*. An integer specifying how many of this connection type are in the 'original' solution.

- *ConnectionMet.* A bool that asks if a player has already met the connection criteria defined in the list item (there are no more of this connection type that have yet to be made), defaulted to be set to false.

Scoring a player's functional connections works in a similar way to scoring using the *FormSolutions* list. However, a key difference is that scoring in this method must also be able access instances of the *Object Controller* class belonging to not just the *selected* puzzle piece, but also the *connected* puzzle pieces in the scene.

When scoring functional connections, the *Scoring* class will, for each *ConnectedInfo* item in the *ConnectedInfo* list in the *selected* puzzle piece's *Object Controller* script component, go through the *FunctionSolutions* list, checking for items where all three of the following are true:

- The *Function* in the *FunctionSolutions* list item matches either the *Function* assigned to the *selected* puzzle piece or the *Function* saved to the *ConnectedInfo* list item saved to the puzzle piece's *Object Controller* script.
- The *ConnectedFunction* matches either the puzzle piece's *Function* or the connected puzzle piece's *Function* (similar to above).
- *connectionsMet* bool is set to false.

If all three statements above are true, then the player's score is increased by one. Since each connection is defined twice in the list, the duplicate is recognized as being correct, however, it does not add to the score and the *correctConnection* bool in both the *selected* puzzle piece list item and its corresponding list item in the *place* puzzle piece is set to true. The *NumRemaining* for that criteria item is then reduced by one. If the *NumRemaining* equals zero, then the bool *ConnectionsMet* is set to true. In addition, no matter if the connection is correct or not, the *ConnectedInfo* item's *hasBeenScored* bool is set to true and the corresponding *ConnectedInfo* items in the *connected* puzzle pieces also have their *hasBeenScored* bool set to true as well.

This process can also happen in reverse, same as in scoring done with the *FormSolutions* criteria list, where the score is subtracted from if a puzzle piece is destroyed/designated to be destroyed which had one or more correct connections.

5.8 Design Rules and Schema

Each completed core action cycle is saved as a *design rule. Design rules* are then saved to a list of *design rules* which are compiled into a list of *design schema*. The *design rules* and *schema* are used to track the choices a player makes as they solve a puzzle level. They are also used for player actions like *exploration* (detailed in section 5.10) and undoing a previous action cycle (section 5.9.2). In this section I will detail out how the *design rules* are created and saved as well as the process of creating the *design schema* list in greater detail.

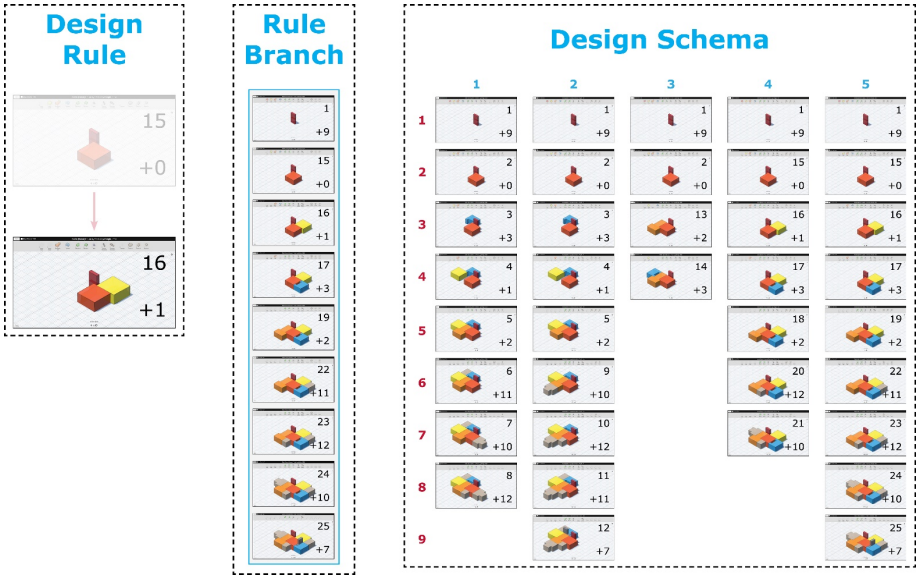


Figure 5-33 Design Rule, Branch, & Schema Overview

Source: Author

5.8.1 Design Rules

At the end of each core action cycle the game saves the choices the player made during that action cycle as a design rule. Because the *design rules* need to be able to be saved outside of a given gameplay session, all of the data it stores is formatted in variables that are not *Unity* specific but can be converted back to *Unity*

specific variables if need be. Each *design rule* saves the following information from the completed core action cycle:

- *Puzzle Piece Info.* An instance of the custom *Info* class belonging to the *selected* puzzle piece's *Object Controller* component, see section 5.2.1
- *Connected.* The *ConnectedPuzzlePieces* list saved to the *selected* puzzle piece's *Object Controller* component, see section 5.6
- *Correct position*. A bool that looks if the player placed the *selected* puzzle piece in the correct position and set up during scoring.
- *Correct Rotation*. A bool that looks if the player placed the *selected* puzzle piece at the correct rotation and set up during scoring.
- *Location.* A custom constructor that takes a Vector3 and a Quaternion as its parameters and translates them into variables not specific to *Unity.* Is used to save a *transform.position* and *transform.rotation*.
- *Player Undo*? – bool that designates if the player has/is undoing this specific design rule. Used for setting up schema branches.

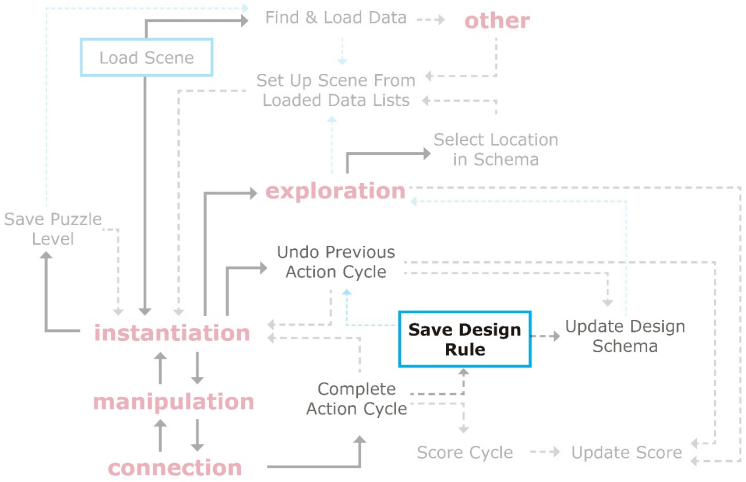


Figure 5-34 Saving Design Rule's Location in Action Cycle

Source: Author

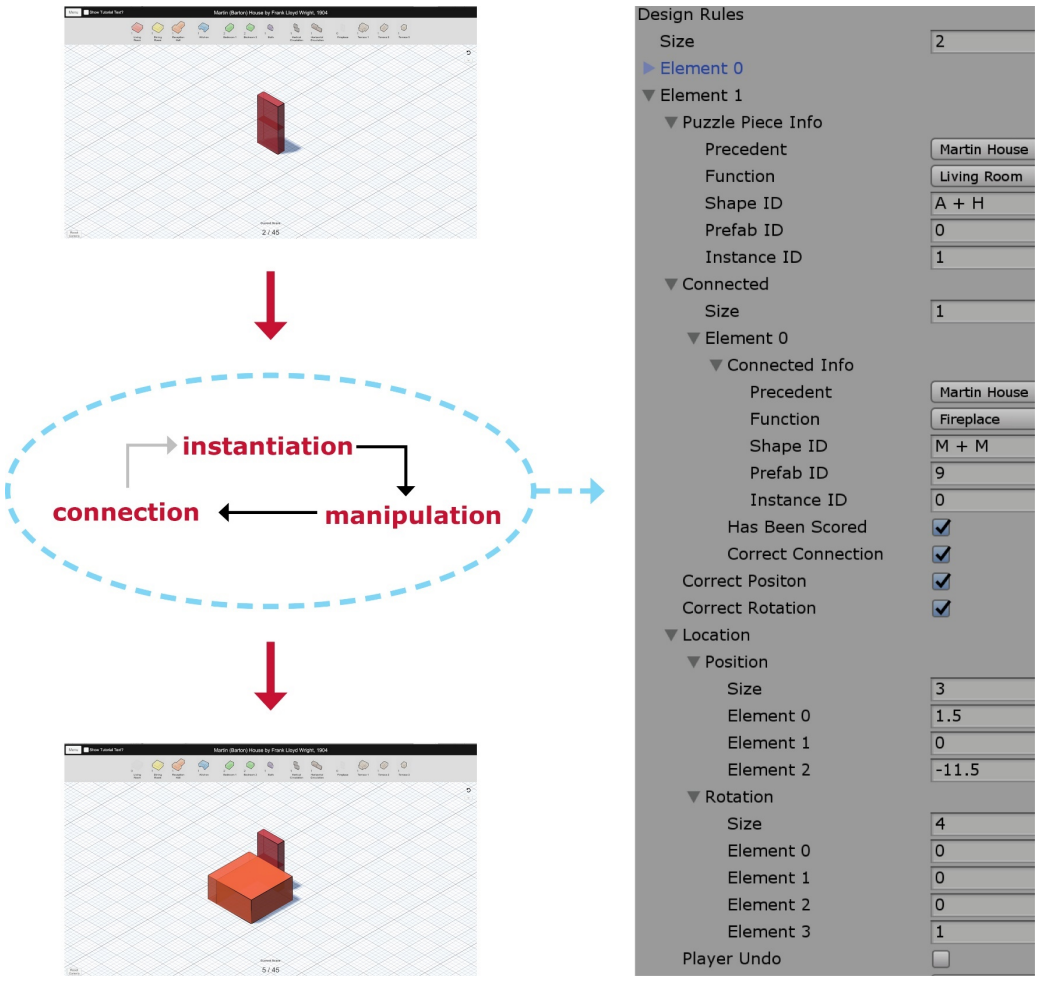


Figure 5-35 Design Rule Implementation

Source: Author

5.8.2 Design Schema

The *design schema* is the overall list of *design rule* lists that communicate the branching paths and choices a player makes, accounting for player backtracking. Each time a player backtracks, by undoing an entire action cycle for example, the game recognizes that, tracks the *design rules* belonging to each puzzle piece destroyed/undone this way before a player opts to instantiate a new puzzle piece in the scene. At that time the game will save a new branch of *design rules* to the *design schema* list, containing all *design rules* not undone by the player (uses the *player undo* bool to determine which *rules* to save to new branch) and is set to the current schema index and is where all new *design rules* will be saved until such time when a player back tracks again. A similar process is also used when *exploration* action cycle which is detailed in section 5.10**.**

What if a player wanted to backtrack, could not just save rules into a single continuous list, which is what had been done in the prior *Unity* experiment/test outlined in section 2.1.

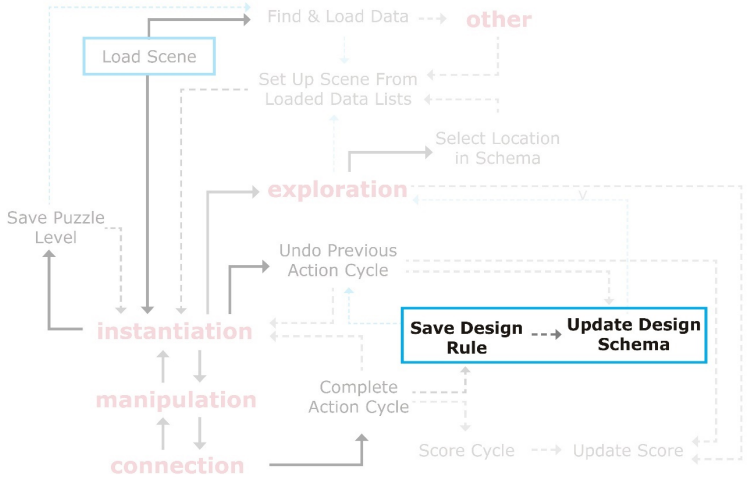


Figure 5-36 Updating Design Schema's Location in Action Cycle

Source: Author

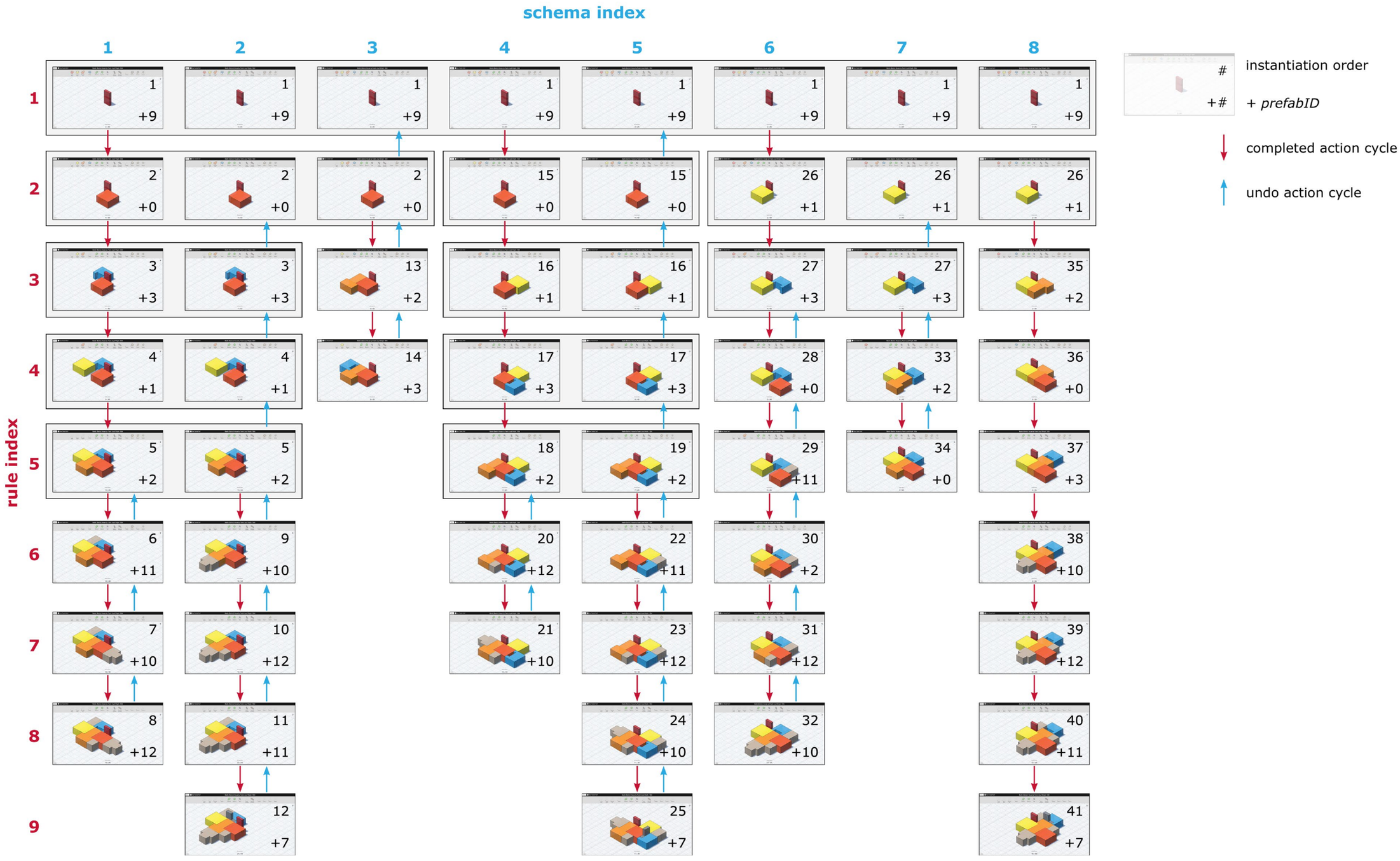


Figure 5-4 Schema Structure

Source: Author

5.9 Player Undo

Going through the puzzles on my own during the development process, I quickly discovered that it would become necessary to allow the player to back out of some of their choices as it is easy to make a small mistake both during the individual steps within an action cycle, as well as whole steps. This subsection discusses the features that touch on how to allow for the player to correct mistakes they make while progressing through the level without necessitating the player restart the level in its entirety. There are currently two types of *undo* options and each are only used during specific stages of the action-cycle. The two types are:

- Step within Current Action-Cycle
- Previous Action-Cycle

The following two subsections will go over both types of undo in greater detail.

5.9.1 Undo within Current Action Cycle

When the player opts to undo while in either the *manipulation* or *connection* action-cycle state the game will take them back to the previous state within the current action-cycle. This means that if the player goes to undo during the *connection* state they will be taken back to the *manipulation* state, and if they are in the *manipulation* state they will be taken back to the *instantiation* state.

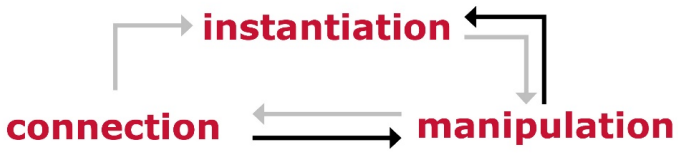


Figure 5-38 Mid-Cycle Undo

Source: Author

This type of undo action is the more simple of the two and can be called with 'escape' key on the player's keyboard or by using the undo button located at the top right of the game window.

If the player calls the undo method while in the *connection* action cycle state, then the game will go back to the *manipulation* state. This transition involves, all

items in the *ConnectedPuzzlePieces* list saved to the *selected* puzzle piece's *Object Controller* script and the corresponding list item in the *connected* puzzle pieces' own *ConnectedPuzzlePieces* lists are removed. Then all *connected* puzzle pieces have their puzzle piece state set back to *placed* and the *Object Mover* script in the *selected* puzzle piece is re-enabled, allowing the player to manipulate its *transform* again.

If the player calls the undo method while in the *manipulation* action cycle state, then the game will go back to the *instantiation* state. This transition is more simple that than the previous one, and only involves destroying the *selected* puzzle piece game object and updating its corresponding buttons so that it recognizes that the player will need to re-instantiate that puzzle piece at a later time.

This type of undo cannot be called during the *instantiation* action cycle state. If the undo method is called then, the *Game Controller* script will call the version of undo which undoes the previous action cycle detailed in the following sub-section.

5.9.2 Undo Previous Action Cycle

If a player calls the undo method while in the *instantiation* action cycle state, then the game will undo the action cycle belonging to the last *design rules* list item whose *player undo* bool is set to false. This allows for the player to undo multiple action cycles without the game losing its position within the *design rules* list found in the *design schema.*

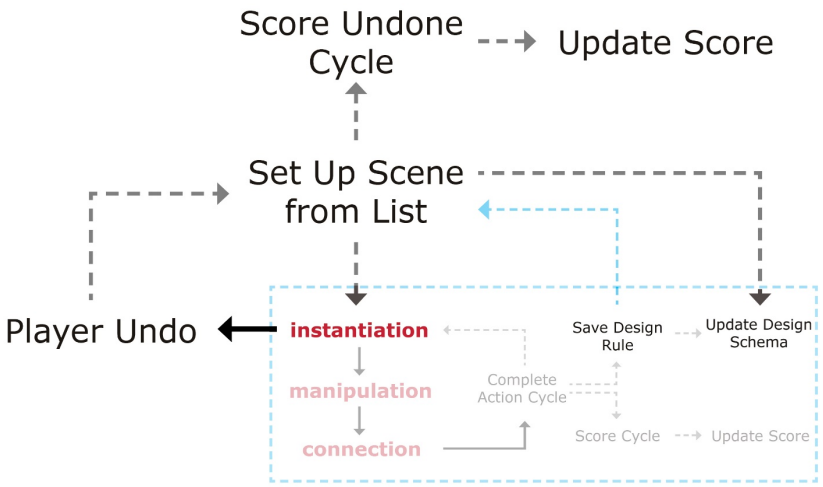


Figure 5-39 Undo Previous Action Cycle Overview

Source: Author

When a player performs this type of undo, all the same actions detailed for undo within a current action cycle are done with the addition of subtracting from the player's score for any scoring criteria met in the undone action cycle. The scoring criteria lists are also updated to reflect that those criteria are no longer met within the scene.

5.10 Exploration

This action cycle state is used when the player explores the *design schema* generated from their actions up till that point. This action cycle state, as mentioned in the previous chapter, can only be entered from the *instantiation* state and exits out to that same state.

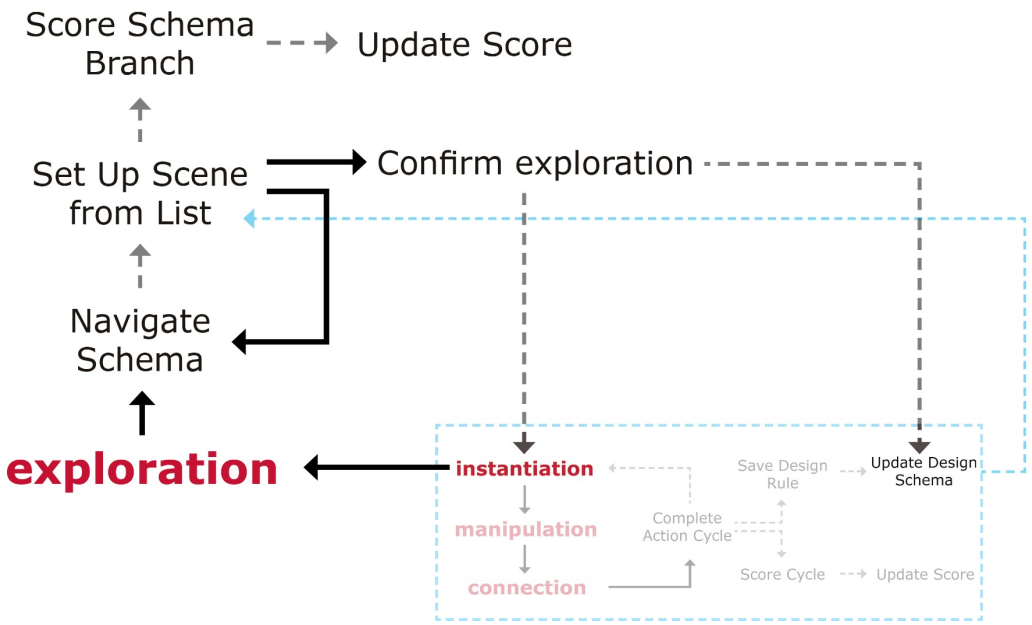


Figure 5-40 Exploration Action Cycle Overview

Source: Author

During *exploration* the player will designate a position within the *design schema* they want to look at, both in terms of the *design schema index* and the *design rules index*. Upon changing the values for either index, the scene is updated to reflect such change, both in terms of the puzzle pieces in the scene and the score. Any puzzle piece in the scene that is does not have a match in the *design rules* list from indexes zero to an integer set by the player at an index in the *design schema*

also set by the player, has its puzzle piece state set to *subtract.* Any puzzle pieces that are instantiated within the scene during the *exploration* action cycle state have their puzzle piece state set to *add.* These puzzle pieces also are instantiated with their *Object Mover* script component disabled so that the player does not move them around the scene. The same also goes for any puzzle piece game objects already instantiated within the scene, but whose puzzle piece state is set to *subtract*.

Once the player sets both the *schema index* and *design rules index,* they want, they can confirm their choice and exit the *exploration* state, starting a new action cycle in the *instantiation* state. Any puzzle pieces whose puzzle piece state is set to *add* has its state set to *placed*, and any puzzle piece whose state is set to *subtract* is destroyed at this time.

If the player chose the last *design rules index* of the branch belonging to their selected *design schema index*, the game controller saves the *schema index* and the branch at that index is where all new *design rules* will be added until the player backtracks again. Otherwise the game saves all the *design rules* up to the selected *design rules index* to a new item in the *design schema.*

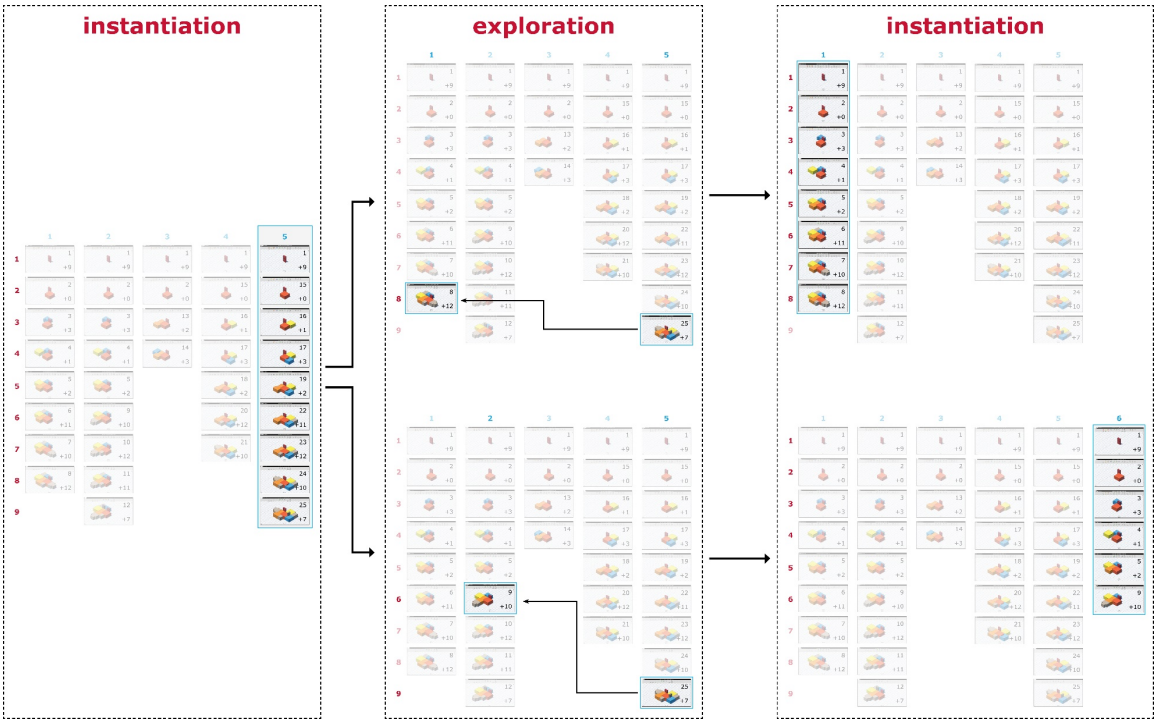


Figure 5-41 Updating Schema in Exploration State

Source: Author

5.11 Camera Controls

Instead of having one fixed view, the player is able to adjust settings within the camera game object to navigate through the scene and get different views of their work. This ability to navigate the 3D space helps the player to visualize and see what it is that they are doing.

Current camera controls have the player adjusting the camera's position, rotation about the y-axis (the axis pointing up) and zoom. As there is no player avatar within the game, the camera servers that role in essence and therefore uses input controls typically associated in pc video games with player movement. Camera position, for example, is tied to the WASD keys and camera rotation is tied to the Q and E keys. Zoom is controlled with the mouse scroll wheel. The choice of using these inputs is that they are common and pretty much ubiquitous in PC gaming, meaning fewer new things for the player to learn in terms of how to navigate the game space.

---

[1] "Transform."

[2] "LineRenderer," Manual (Webpage), Unity Technologies, updated May 31, 2017, accessed March 11, 2019, https://docs.unity3d.com/Manual/class-LineRenderer.html.

[3] "Transform."

[4] "Transform."

[5] "Object.Instantiate," Scripting API, Unity Technologies, updated March 13, 2019, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/Object.Instantiate.html.

[6] Schön, "The reflective practitioner : how professionals think in action."

[7] Koning and Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses."

Chapter 6: Discussion & Framework for Future Development

The project detailed within this document is an initial iteration of a potential video game which seeks to achieve the following. One, the tracking and analysis of player actions during gameplay as design rules saved within a schema that the player can explore and reuse and two, convey some tacit knowledge of architectural design using architectural precedents and by providing the player with meaningful feedback. While development of this first iteration which comprises this project is over, the development of the proposed game will hopefully see continued development effort elaborating on and resolving ideas and issues which arose during these early stages in addition to adding additional features which would facilitate in helping the game achieve its stated goal.

This chapter outlines some of the gameplay features and aspects of game design and development which were brought out in committee meetings but that, for time and resource reasons, were not implemented within the current build of the game. Some of these things will be new features and elements of game design that have only been mentioned briefly to acknowledge their importance, while others are adjustments to existing features or game design aspects that should be considered in future iterations.

The last sections of this chapter serves as a summary for the whole process up to the end for this dissertation project and where I would like to see it go from there.

6.1 Player Testing

One of the next steps for this project would be to enter player testing. Unfortunately, there was not room within the given timeframe of this project to begin player testing; however, some thought has gone into the general idea of what to look at in the initial round of player testing.

Player testing should look at how hard the puzzles are to solve, if there is anything awkward or unnatural feeling in the player input controls, and if the information being connived via the UI working as intended, what information isn't coming across or is coming across incorrectly. I have been doing most of the testing myself and I know how the correct solution to the Martin House puzzle is put together. Someone new to the game, who has not been as involved in the development process as I am can see the current build with fresh eyes and, hopefully, notice problems I cannot see.

Most of the sections in this chapter will acknowledge that, before adjustments to some of the currently gameplay features are looked at, there should probably be some form of player testing done.

6.2 Framing & In-Game Narrative

While not critical to this initial stage of development, in future iterations it will become increasingly important to consider the framework used to hold together the game's library of precedent puzzles in addition to how the players are introduced to each one. Part of this includes thinking about what kind of stories might be used to guild the player through the game and between the puzzles. What drives their goals within the game's narrative structure? Several possible in-game narratives have been brainstormed, but currently have not seen any further development.

In a similar vain to in-game narratives, is how the game is presented to the player goes beyond simple in-game narratives, how the puzzle levels, the *library of precedent puzzles*, are connected and structured. This structure/connection between the various precedent puzzles within the *library* is the games progression path. So far, two different progression paths have been given initial consideration,.

Following subsections outline what work has currently gone into looking at developing the overall structure and narrative of the proposed game, however, much work still needs to be done fleshing these ideas out for implementation into future game iterations.

6.2.1 In-Game Narrative

Narratives in games in video games are more than just the plot of the game. The audio, scene design, plot, player choices, and gameplay all work together to create and communicate an overall experience that can draw a player into a story.[1] The scope of this dissertation project did not allow for the development of what a narrative for the proposed video game would be, but some initial brainstorming for what that narrative might look like was done after the selection of scenario 3.1 (see section 2.5). The following table outlines five potential narratives by defining a setting where the game would take place, who the player is playing as, and what the player character's goal is within the narrative.

1	**Setting**	University
	Player	Architecture Student
	Goal	Learn about architectural precedents
2	**Setting**	Architectural offices
	Player	Time-traveling architectural intern
	Goal	Assist architects through-out history in designing historical works of architecture
3	**Setting**	Post-Apocalypse
	Player	Descendent of a group of survivors
	Goal	Rebuild famous landmark buildings
4	**Setting**	Distant Future
	Player	Historian
	Goal	Analyze important historical structures from the past
5	**Setting**	Outer space
	Player	Alien
	Goal	Analyze human architectural artifacts

Figure 6-1 In-Game Framing Narrative Brainstorming
Source: Author

6.2.2 Progression Path

The progression path of the game refers to how a player would move between puzzles. This affects the order of the puzzles, which puzzles the player solves when they first start playing and what puzzles come after. Two different types of progression paths are detailed in this section: the *puzzle book* and the *narrative thread*. Each come with positives and negatives.

Puzzle book is the progression path which most directly resembles the *library of architectural precedents* idea detailed in section 4.2. In this progression path a player selects specific puzzles to solve from several nesting lists. They can choose to solve the puzzles in the order they are presented or not. The pros to this type of progression path is that it provides the player with more agency by allowing them to explore the puzzles they want in the order they want. This also means that if a player gets stuck on one puzzle, they could simply work on a different puzzle. The cons of this are that it reduces the control the developer has in creating a difficulty

curve and introducing new aspects of gameplay to help ease a player into the experience. [2]

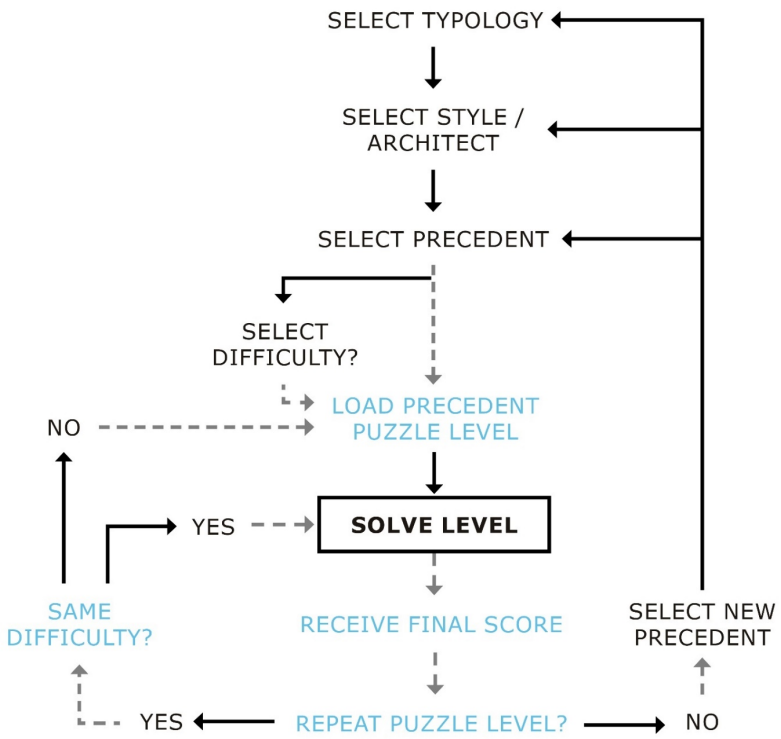


Figure 6-2 Player Progress - Puzzle Book

Source: Author

Narrative Thread is the other progression path that was roughly outlined in which the player is guided along a story where at each story beat, they would solve a precedent puzzle. The positives of this kind of path is that there is more control on the developer end to help tailor the player experience and slowly introduce new and more complicated puzzles and gameplay mechanics. The downside is that the player has less agency and can end up stuck on a puzzle level and, since the puzzles are solved in a particular order, they have no way of progressing in the game.

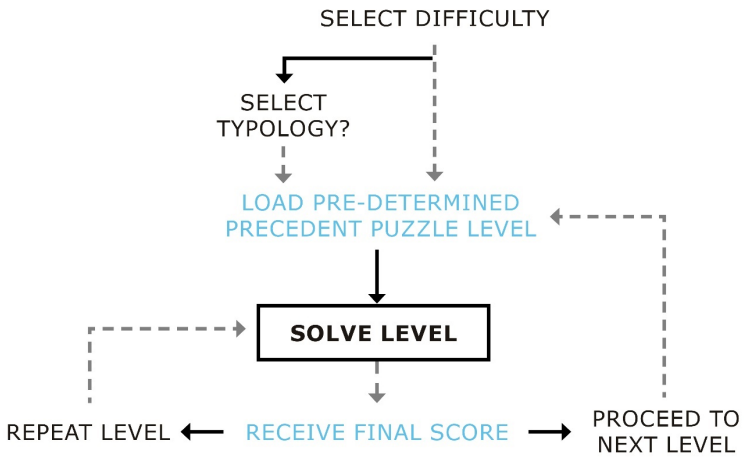


Figure 6-3 Player Progression - Narrative Thread

Source: Author.

6.3 Expanding Library of Precedent Puzzles

A core goal laid out at the very beginning was for this game to contain a *library of precedent puzzles* for the player to explore and solve. Currently the Martin (Barton) House has been converted into its corresponding puzzle and the Villa Savoye has already been abstracted into its puzzle pieces and only needs to be assembled within the *Unity* editor. However, a game containing only one or two geometric puzzles is not likely to remain interesting for very long and having only one or two books does not make for a very good library. Therefore, additional architectural precedents will need to be selected and abstracted. Once abstractions exist for these new precedents, the corresponding puzzle levels will need to be assembled. While all of this sounds fairly straightforward, how the puzzle levels themselves are assembled within the overall structure of the game needs to be taken into consideration. This ties directly back to the selecting of a framing device discussed in the previous section.

Ideally the end game product will contain puzzles from a wide variety of architectural styles, time periods, scales, and typologies. There is still a lot of work ahead in terms of both selecting and abstracting precedents for implementation of puzzle levels. This process of adding onto the *library of precedent puzzles* is also something that could take place after the game is released, where bundles of these precedent puzzles could be made available as dlc (downloadable content).

6.4 Information Provided to Player

In Section 4.11.1 three main types of information that can be conveyed to the player about the precedent puzzle they are solving is outlined. These three types of information are the following:

- *Embedded Information.* Information embedded within the puzzle pieces themselves, like having them already be labeled when the player starts the level.
- *Contextual Information.* Images and information about the precedent shown to the player either prior to them going through process of solving said precedent's puzzle.
- *In-Game Hints.* Information contained within the scene which could help guide players if they get lost.

All three types have been looked at, with the first, *embedded information*, seeing some implementation within the current game prototype. However, even that one has room for improvement for potential implementation in future iterations of this game. The following subsections will outline some of our thoughts about directions to go for conveying information to the player. The ultimate goal of this type of information is, similar to scoring feedback, help guide the player and give them a sense of direction for how to solve the different puzzles

6.4.1 Embedded Information

Embedded information is any information conveyed through the core parts of the game that the player is interacting with during the core action cycle. It is *embedded* into the core gameplay mechanics. three ways for improving information provided to players through *embedded information* were brought during the development of the prototype build of this game. The first two will deal primarily with the *instantiation* action cycle state and the third with the *connection* action cycle state.

The first looks at what adjustments can be made to the instantiation UI panel. Players currently have no way of knowing what floor level each puzzle piece is assigned to prior to instantiating them as this information just is not communicated in any of the UI used. There has also been some confusion where someone has mistaken the textural information meant to convey the number of a particular puzzle

piece was instead referencing the floor level of the puzzle piece. Any new iteration should address this when setting up the instantiation UI panel.

The second place for improvement has to do with the instantiation process itself. While the images of the puzzle pieces in the instantiation buttons is nice, it might also bee nice if the player was able to preview the item in the scene before committing to its selection. It would be nice to see what the puzzle piece actually looks like in relation to the pieces already in the scene. This would ideally happen when a player's mouse hovers over the instantiation button.

The third area for improvement has to do with how the game conveys to the player which puzzle pieces are connected or not. Currently this information is only conveyed to the player during that specific puzzle piece's action cycle. This means that once the player has completed an action cycle, they have no way of going back and seeing how they connected it or any other puzzle piece in the scene. This is a major problem that needs to be addressed. It is currently unclear how such information might be expressed and solutions need to be brainstormed.

6.4.2 Contextual Information

Figure 6-4 Location of Initial Information Provided to Player Within Puzzle Level Loading
Source: Author

Contextual information is information provided to the player prior to starting the puzzle, giving the puzzle context. It could include information on the main game menu, found in the loading screen for each puzzle, or in a separate scene (maybe even a cut-scene) that runs before the scene containing the puzzle is loaded.

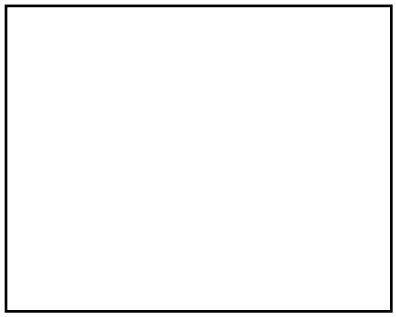


Figure 6-5 Isometric Drawing of Martin House
Source: Koning and Eizenberg

This information could be things like having the player click on a button with a picture or isometric drawing of the precedent behind the puzzle they are about to solve. This information can also be words, spoken or written, about who the architect was/is, where and when was it built, and what makes it important, what is it known for. Some of that information will be more helpful than the others,

but it might also spoil the puzzle if too much information about the design gives away the original solution too easily. All of this information can help build a picture in the player's mind of what they should be aiming for, but if they are given too much information they might not be as willing or able to explore potential variations.

6.4.3 In-Game Hints

Information provided to the player during gameplay of the various puzzle levels has only been explored in a very preliminary capacity. This type of information is more difficult to generate than the other two, as if the information is too precise in its geometric appearance, it could discourage players from creating design variations. On the other hand, if not enough information is conveyed, then the hint is not helpful. Currently the idea for how to convey information about formal and functional layout to the player without being so concrete as to solve the puzzle for them, is to create a kind of 3D visual trail which could convey the overall flow through the precedent spaces and provide very basic functional divisions (between public, private, and service for example). I will refer to this 3D visual effect by several names throughout this project, but I will most frequently call them faerie lights.

Figure 6-6 Abstract 3D Effect Idea
Source: Author

Figure 6-7 Spell Effect from the Elders Scroll V: Skyrim by Bethesda
Source: Bethesda's Skyrim

This idea takes its influence from several in-game visual effects and spells that I have seen in my many years of playing video games. One of these is the clairvoyance spell in Elders Scroll V: Skyrim. This spell, and other abilities similar

from other game titles, is thematically trying to do something similar, lead a player to an end destination. How these faerie lights would work is that they would highlight the placement and basic color indication for where the "heart" or "core" of the precedent is located, then having several of these faerie lights trail out from that center, each belonging to one of the basic functional zones (public, private, and service in the case of the Martin House). These trails would roughly sketch out the overall size allotted to each of these zones. These faerie lights could also change location based upon where the player locates/places the object that is predesignated to be the "core."

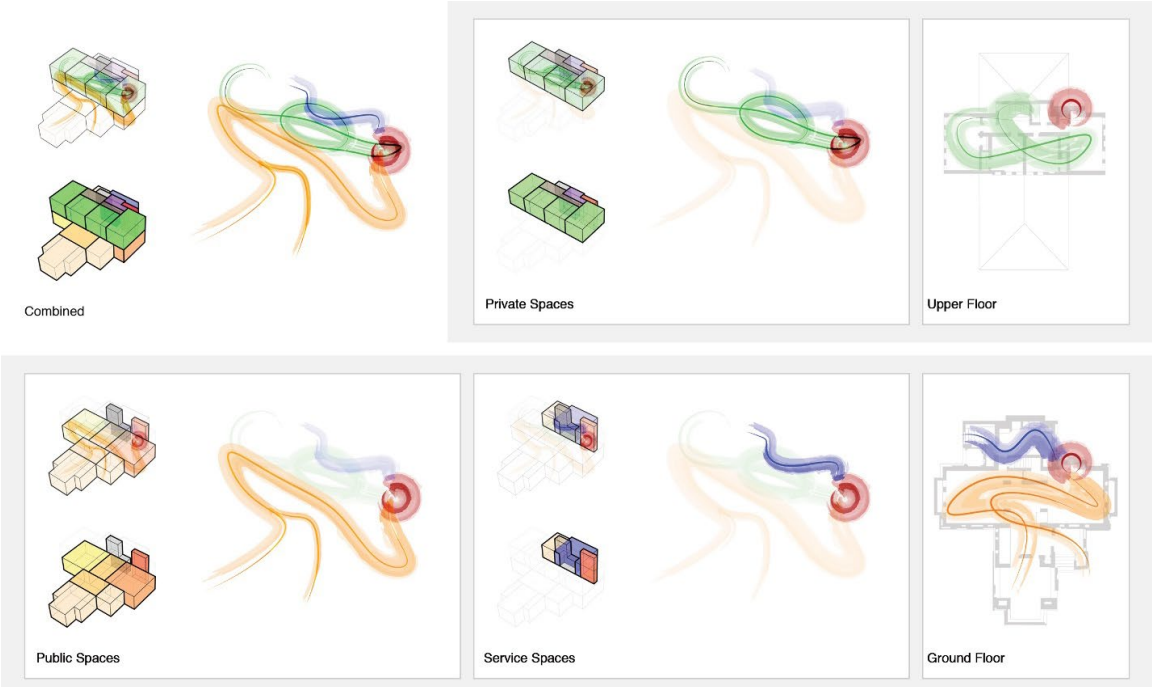


Figure 6-8 Faerie Lights (Abstract 3D Effect Hint) Example for the Martin House (Barton House)

Source: Author

Another thing to consider as the faerie lights relate to gameplay, is whether or not the effect is always on. Are these trails always visible in their entirety? Or, does the visual effect move down the trail, away from the "core" when the player asks for a hint?

This idea is untested within the current build of the game, and would need to be implemented in some capacity before any conclusions can really be drawn about its usefulness and feasibility.

6.5 Player Difficulty Settings

Player difficulty has been examined through a series of gameplay mechanic variables detailed below. It is important to note that only one of each option has been implemented in this iteration of the game, with the aim of implanting the remaining options in future development iterations. Some of the more obvious of these variables that have come up in the process of abstracting and testing precedent designs are:

- *Room Functions*: is the player responsible for *naming* the shapes they are using (do they already have their original room function assigned to them when the player instantiates them)?
- *Compound Shapes*: is the player provided with compound shapes from the start, or do they have to create them themselves, being only provided with a set of basic and unique shapes to work with?
- *Floor Levels*: Is the player told what rooms belong to what floor?
- *Placement Constraints*: How free is a player when it comes to placing and moving shapes within the scene? What does the game look at/for when a player wants to move a shape?
- *Hints*: How much information is the player given in regards to the precedent's originals design"

The figure below goes over options of for how these different variables can adjust gameplay difficulty, ranging from a gameplay experience with more hand-holding elements to one where the player is pretty much on their own in terms of figuring out what to do. It is important to note that as player freedom goes up, so to does difficulty.

	ROOM FUNCTIONS	COMPOUND SHAPES	FLOOR LEVELS	PLACEMENT CONSTRAINTS	HINTS
Casual	Predefined - specific rooms	Pre-grouped	Shapes pre-assigned to correct floor level	Corner-to-corner, connection-to-connection	Isometric building image, Faerie-lights on by default
Easy	Predefined - specific rooms	Pre-grouped	Player assigned	Corner-to-corner	Isometric building image, Faerie-lights on by default
Medium	Predefined - general uses (public, service, private, etc.)	Pre-grouped	Player assigned	Surface-to-surface	Isometric building image, Faerie-lights on by player request
Hard	Player defined	Pre-grouped	Player assigned	Surface-to-surface	Isometric building image, Faerie-lights on by player request
Very Hard	Player defined	Pre-grouped	Player assigned	Increments based on precedent	Isometric building image
Free-Form	Player defined	Player grouped	Player assigned	1/2′ & 15º increments	None

Figure 6-9 Difficulty Setting Options

Along the same lines, the process of how a player would choose and adjust their gameplay experiences in relation to difficulty, have only been looked at in terms of the potential for future development. The difficulty settings and process of difficulty selection from other games have been examined in this capacity, like the example shown below form Pillars of Eternity II: Deadfire by Obsidian. In this game overall difficulty options are selected whenever a player starts a new game with further minor tweaks that the player can make from within the in-game menu during gameplay.

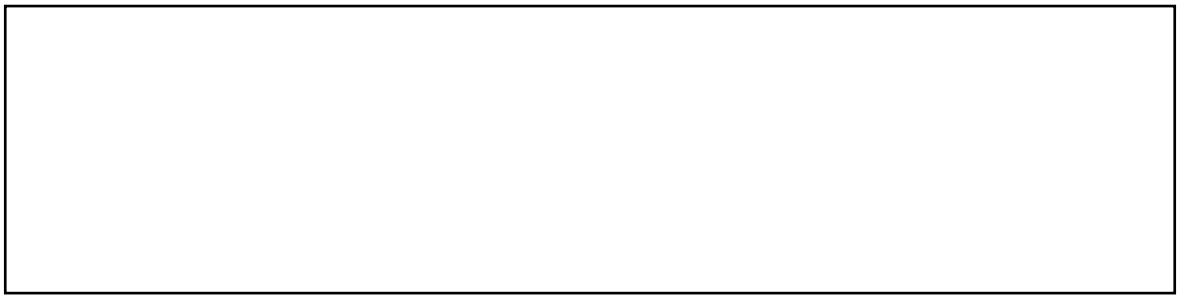


Figure 6-10 Examples of Difficulty Settings from Pillars of Eternity II: Deadfire by Obsidian
Source: Obsidian

Implementing player difficulty options will require some additional infrastructure to be added to the game as the game will need to both remember a player defined difficulty choice and adjust existing methods and UI settings based on those choices.

6.6 Manipulation Constraints

Within the current build of the game, the manipulation action cycle state seems to work pretty well. There are some awkward things with the controls where rotating puzzle pieces sometimes does not work fully as intended and moving second floor puzzle pieces feels very awkward as the position is based off where the mouse curser is in relation to the ground plane. These are annoying but, ultimately minor bugs that still need to be ironed out. Outside of the not fully fleshed out player input controls, additional aspects and behaviors for this state in the action cycle have been discussed throughout this project, but did not have time to be fully fleshed out and implemented. This section outlines those thoughts so hopefully they can be used in future iterations.

As mentioned in the previous section on player difficulty settings, different types of *manipulation* contestants could be a useful thing for the proposed game to have. Only corner-to-corner snapping has seen full implementation at present. This section looks at a possible expansion to the corner-to-corner snapping system, an idea for the addition of surface-to-surface snapping constraint, and an overall constraint for the *manipulation* state as a whole by looking for puzzle piece overlap.

6.6.1 Corner-to-Corner Snapping

Before getting into the additional placement constraint options, there is an unanswered concern that has been brought up about corner-to-corner snapping. Namely whether or not to allow players to place a shape in such a way that it only shares an edge with the initial shape within the scene. Spaces like the one's shown

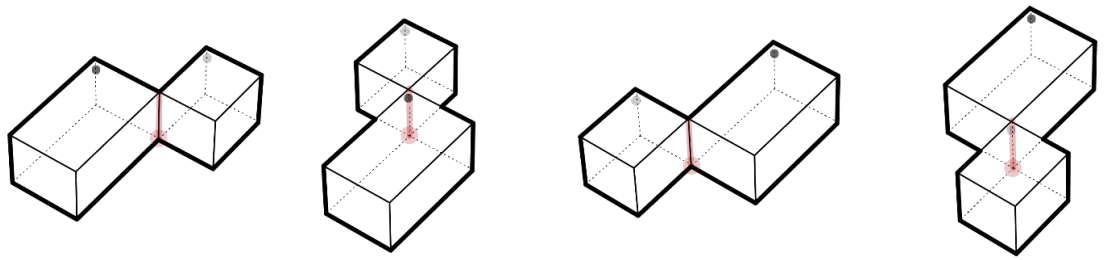


Figure 6-11 Edge Overlapping without Touching Surfaces

Source: Author

below do not make much sense architecturally as you cannot really get from one space to the other. In order to prevent this the game would need to be able to recognize when and adjust the selected shapes location so that they share an

overlapped surface in addition to having one or more of each shapes' corner points aligned. The game would need to recognize shared vertical and horizontal surfaces.

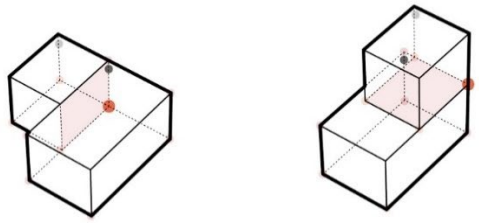


Figure 6-12 Snapping at Same Plan Level (left) & at Adjacent Plan Levels (right)

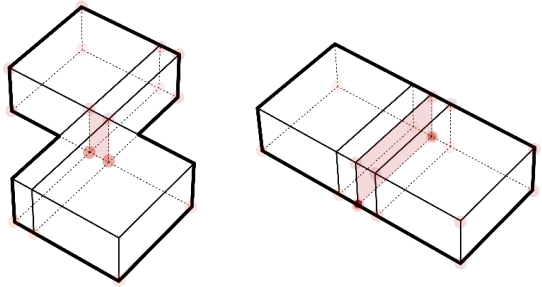


Figure 6-13 Touching Surfaces
Source: Author

6.6.2 Surface-to-Surface Snapping

As mentioned very briefly in the previous section dealing with player difficulty settings, addition ways of snapping and constraining puzzle piece *manipulation* is a good way to let the player have more of a say in how much freedom they want, with the understanding that sometimes the more freedom a player has, the harder it is for them to arrive at a satisfactory result.

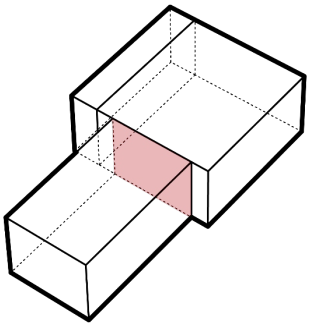


Figure 6-14 Translation along Snapped Surface
Source: Author

One of the other *manipulation* constraints that was thought up around the same time as corner-to-corner snapping is surface-to-surface snapping. Surface-to-surface snapping is almost

ready for implementation, as should work very similarly to corner-to-corner snapping. Corner-to-corner snapping works through a series of trigger colliders located at the corners of each puzzle piece, that when triggered by another corner trigger collider, moves the selected puzzle piece so that their corner align and locks their x, y, and z *transform.position* values until the player moves their mouse far a certain distance away. Surface-to-surface would work exactly the same except instead of restricting all three x, y, and z coordinates in the puzzle piece's *transform.position*, it would lock the either x or z coordinate (the y being locked based on the floor level the puzzle piece is assigned to), allowing the player to adjust the other, sliding the selected puzzle piece along the snapped surface.

As far as implementing this feature goes, not too much more work needs to be done. The colliders currently used for *connection* could potentially also be used here, serving two purposes. A means of snapping to puzzle pieces on other floor levels will need to be developed though, as the *connection* trigger colliders are only on the side surfaces, and do not interact with anything on other floor levels. Aside from that the next step towards implementing this feature is creating either a method or adjusting the existing translation method to lock movement in either the x or z coordinate while still allowing movement in the other. A method for unsnapping the surface would also need to be written.

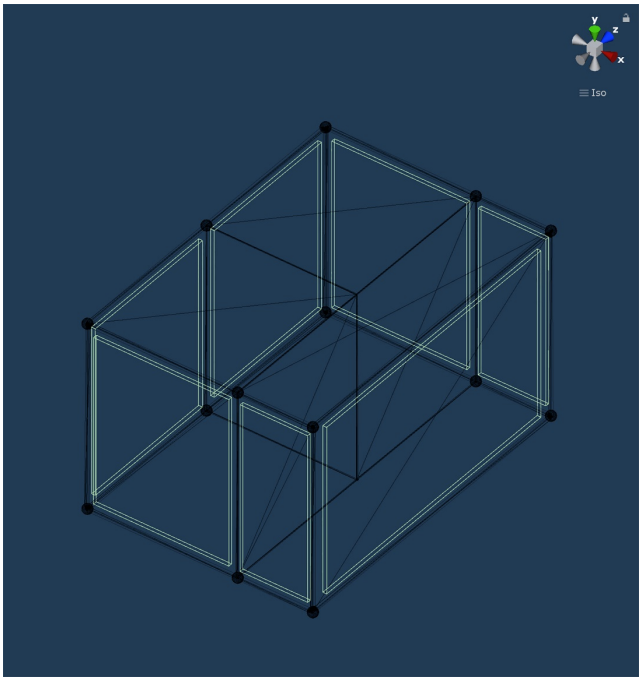


Figure 6-15 SrfConnects in Current Terrace 2 Puzzle Piece

Source: Author

The initial implementation of these methods and behaviors can and should be done prior to working too much on figuring out snapping triggers for puzzle pieces on other floor levels. Once those behaviors are added to the existing code of the game, the developer will need to check that the *connection* trigger

colliders reach out far enough away from their puzzle piece to be useful as they hug pretty tight to the puzzle piece's sides.

6.6.3 Overlap Detection

The current iteration of the game has no way to prevent players from accidently placing their selected shape within a shape in the scene. This becomes particularly relevant when unique and unique compound shapes are involved as they contain non right angles and curves.

Colliders currently exist within the puzzle piece game objects that could be used in this feature's implementation, all they are missing is some additional coding to recognize when two or more puzzle pieces overlap, prevent the player from progressing to the *connection* action cycle state, and visually highlight the overlapping puzzle pieces for the player. This feature has been successfully implemented in the prior work done in *Unity* and should be fairly easy to adopt to the current game, however, as this game is more complex and manipulation of game objects is different, adjustments to the method used before will be necessary.

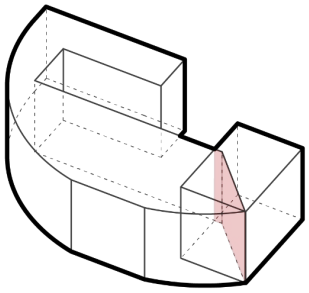


Figure 6-16 Overlapping of Unique Shape
Source: Author

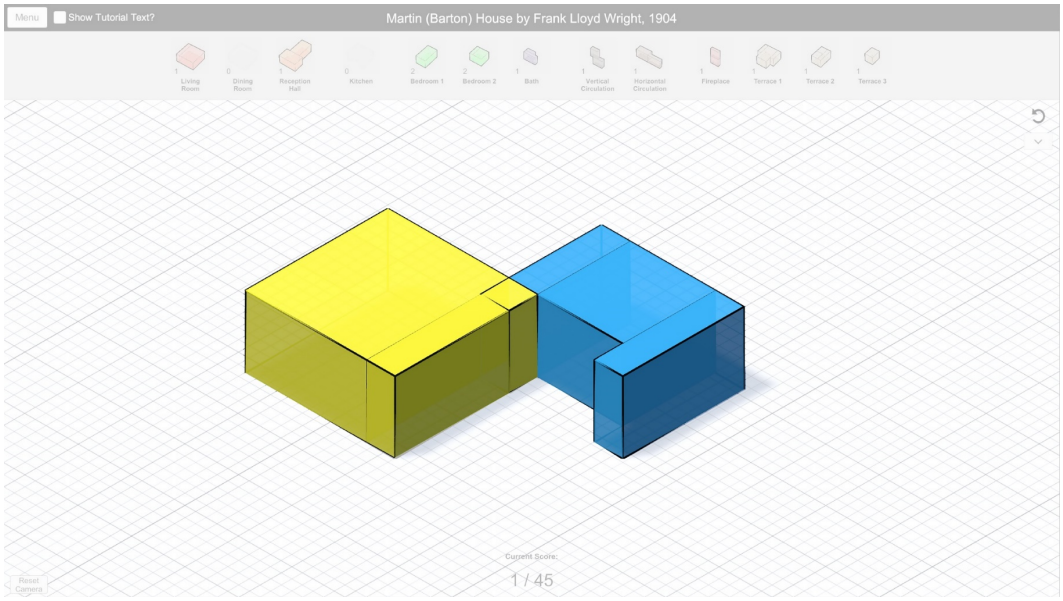


Figure 6-17 Puzzle Piece Overlap in Current Build
Source: Author

6.7 Scoring

In a similar vain to the idea of introducing a "creative play mode" it might be worth it to reintroduce one of the scoring criteria that was cut from this iteration of the game due to time constraints. This additional criterion would be accuracy of design logic and would look at the overall design principles that govern the design as a whole, not just the specific formal aspect of the original.

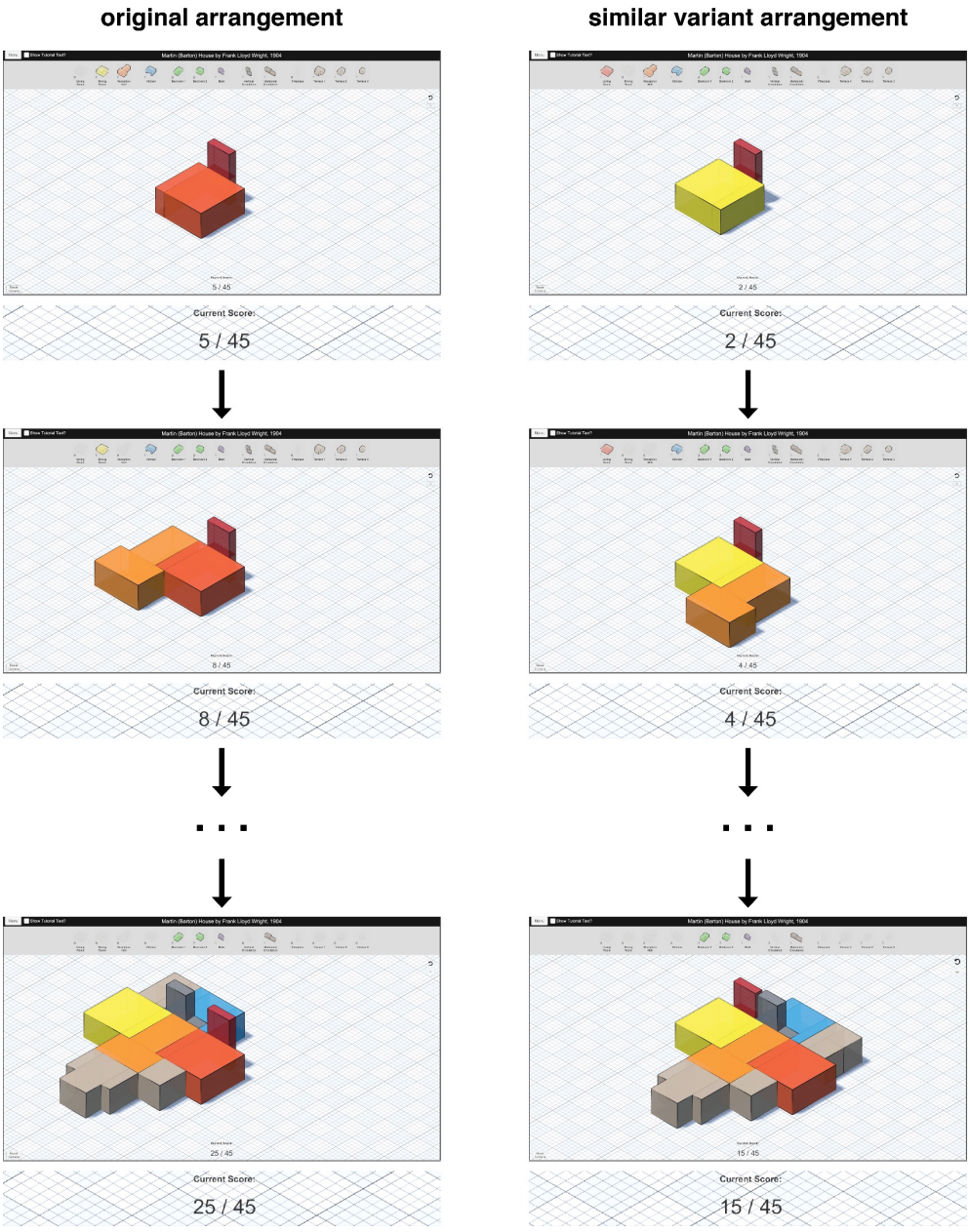


Figure 6-18 Scoring Variant with Minor Changes from Original

Source: Author

This addition to scoring comes into sharp relief when dealing with the scoring of player solution's design arrangement which is based entirely off the first puzzle piece they place within the scene. The reason behind that decision was so that a player's score did not take an unreasonably big hit if the player simply did not pick the correct *player initial object*, however, the game does not recognize when the player follows the overall logic or smaller scale spatial arrangements if they are incorrect in relation to the *player initial object.*

Adjustments to the weight each criteria has is also necessary. Currently the player, upon completing a player action cycle, gets one point for choosing the correct rotation, one point for choosing the correct position, and one point for each correct connection made. These values, as well as any values assigned for meeting any new criteria added, will need to be tested and adjusted to determine what value works best.

Other methods of communicating scoring information that are not quite as direct as showing how many points out of the max possible they have should also be looked into. This can include changes in the lighting, colors, audio, or visual effects. It might also be worthwhile to reevaluate if scoring needs to happen at the conclusion of an action cycle and if it would be possible to update a player's score throughout it instead. Again, all these things will need to be tested continuously through development so it can account for any new features or adjustments to existing gameplay features.

.

6.8 Design Schema

The way the *design schema* currently works in the current build is such that it tracks and saves players' steps as they complete action cycles as *design rules* which are compiled into a list of *design* rules. This list is then saved as a list item within the *design schema list*. The game recognizes when a player does any backtracking of whole action cycles and creates a new *design rules* list which is added to the *design schema* so that the previous *design rules* list does not get over written. In this way the game recognizes where paths diverge, but it does not currently look for convergence where the player takes different paths but still ends up

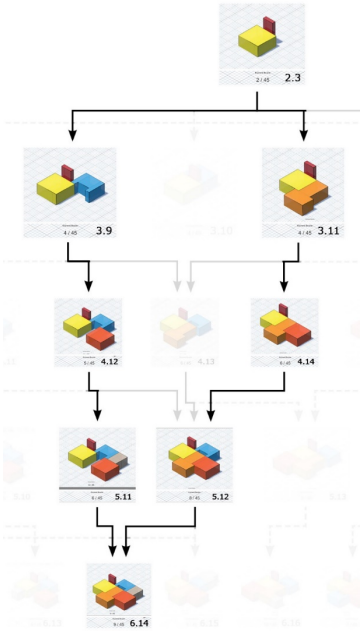


Figure 6-19 Convergence
Source: Author

at the same place. While this feature does seem interesting on the developer/researcher end, it is unclear how often convergence would actually occur during the gameplay of a single player.

Player testing would need to be done to see how they use the current *design rules* and *schema* system before any major overhauls are done. It is worth noting that since the *design schema* was the last feature implemented in this iteration of the game that it has not seen as much testing as other features so it is very likely that the way it has been envisioned being used and what players do with it might be different.

6.9 Player Undo

Currently players can undo decisions and choices they have made at either the individual action cycle state level or at the overall core action cycle level. However, both of these cases only allow players to undo decisions based on the order they made them. It was brought up in the committee meetings that the game should allow players to select an already place puzzle piece and simply delete it, regardless of when in the players process it was placed. Implementing a way for the player to simply delete a puzzle piece in the scene is not a problem, as a method for such a thing already exists and is used when the player undoes something. However, it is not quite as simple as just telling the game to destroy a puzzle piece and then updating the instantiation UI panel to reflect that said puzzle piece will need to be re-instantiated later. The way that the *design rules* and *design schema* currently work does not support doing this outside the reverse order the puzzle pieces were instantiated in.

As mentioned in the design schema section of this chapter, I am hesitant to do any major overhaul of the current *design schema* until some player testing has occurred. I am curious how this restriction on how puzzle pieces can be removed for a scene effects how players play the game and if it might incentivize them to use the *exploration* actions.

6.10 Multiplayer

Given the recent trend in gaming towards the incorporation of multiplayer features within games, initial discussions where had as to how this game could

support this type of gameplay. While no steps have been made towards implanting any kind of multiplayer in existing build of the game, it has been brought up at several times during its development. Based on personal experience, there are several different ways of incorporating multiplayer elements into a game. Some games, like Fortnite or Fallout 76, are multiplayer games where the players occupy a shared game world. In these games player characters exist within the same "game world" as other player characters and are able to directly interact with other player characters.

Other games, like Middle Earth: Shadow of War or the Sims 4, allow players to interact with things from another players game, where the players do not directly interact with each other. For example, in Middle Earth: Shadow of War a player is able to fight and recruit orcs from another players playthrough without effecting the other players game and in the Sims 4 players are able to upload lots, rooms, and families they create in their game to a library where other players can download them to use in their game.

There are several possible ways to begin incorporating multiplayer elements into the proposed game. It could be something where, like in Minecraft, the players occupy the same game world and can work together to solve puzzles which, if the precedent puzzles were bigger in scale than single family housing like say urban planning or high-rises, could add another layer of gameplay. Multiplayer could be direct like the idea stated before but exist in its own gameplay mode. Many single player games coming out have a multiplayer mode, like Dragon Age Inquisition or Minecraft, and many multiplayer games have single player stories like Mortal Combat or the Call of Duty games. Future development could also look at more indirect means of incorporating multiplayer like, for example, allowing players to share their puzzle solutions and the schema for them to other players.

Another dimension to multiplayer how it can be used in both completive and corporative play. Are the player working together to solve a problem, working to best other players, or some combination of the two.

It is worth mentioning that I, the author, do not have a great deal of experience with multiplayer games as they are not something that I am that interested in personally and actively avoid. Therefore, it would be a good idea to bring someone in on the development side who is more experienced in that aspect of video games.

6.11 Gameplay Modes

Different types of players, not everyone is going to care about solving a puzzle for a pre-existing precedent and thinking of ways to expand the potential of the game beyond the core concept of a puzzle game based on architectural precedents. The last section mentioned the potential for a multiplayer gameplay mode, but that is not the only potential mode of gameplay that could implemented. Again looking at Minecraft, a creative build mode where the player is provided an unlimited amount of each object to build with could add a way of engaging with players more interested in the creative aspect a game.

It might also be worth while to look at how a player might import their own puzzle pieces to work with, so they are not limited to just the pieces set up by the developer. They may not even need to import anything, it might be something where they can adjust the *transform.scale* of existing puzzle pieces or regroup existing individual shapes to form their own custom compound shapes/puzzle pieces.

There are also gameplay modes that function like difficulty settings, where the game adds or removes features to make the game dramatically more difficulty, like limiting the number of saves or giving the player a time limit for certain tasks.

6.12 Graphics and Visuals

While some work has gone into tinkering with how to graphically convey information and make the game visually appealing, there is a lot of work that still needs to be done. Many of the UI adjustments to help better communicate information to the player are detailed in section 4.11.1 . These include information about what floor level each puzzle piece belongs to and which puzzle pieces are connected to one another. In addition to addressing those concerns, there are also other graphics centric that need to be tackled to help make the game more visually appealing.

Currently the puzzle pieces use various levels of transparency to convey information specific to them and the actions that can be performed on them based on the current action cycle state. The screenshot above is during the *instantiation* state and, as can be seen in the image, it looks like something very bad is happening with the transparency in the horizontal circulation puzzle piece (the dark brown one). This glitch needs to be fixed.

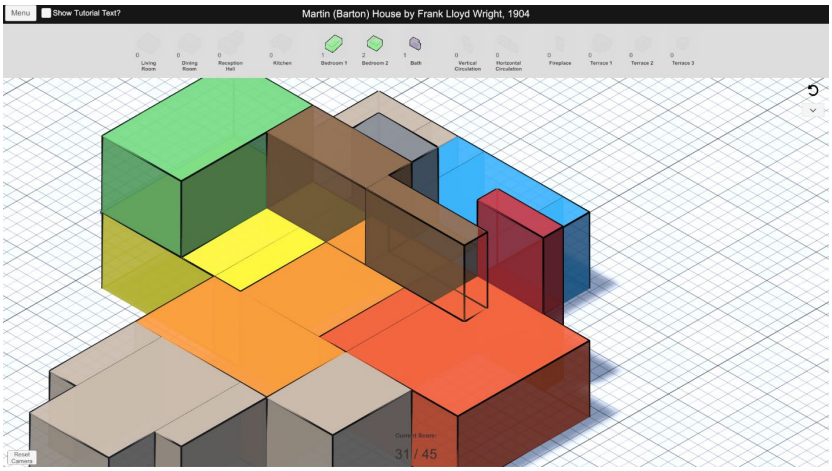


Figure 6-20 Transparency Glitch

Source: Author

Once that transparency glitch has been corrected, there are still other adjustments that can be made to the existing graphics in addition to looking into subtle visual effects that could greatly improve the overall look and feel of the game. It is important to remember that a lot of the information about the puzzle pieces and what actions the player is supposed to do and to what they are supposed to be doing them is conveyed through the visuals, through transparency, color, and lineweight. The less information that is communicated solely through text the better. This is not saying that no text is the goal, but rather that if the player is able to "read" information from more than one place, and that information is embedded into the look and feel of the game, it should help make the game feel better to the player.

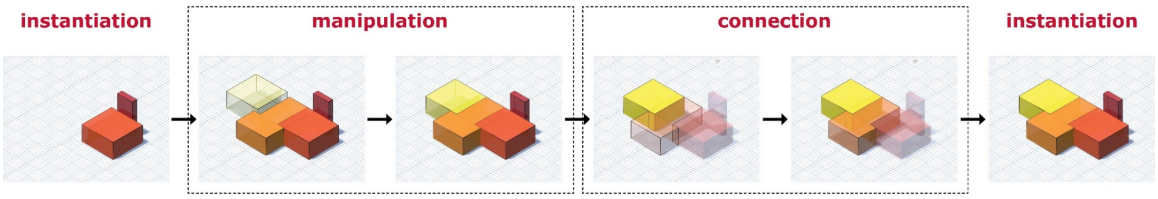


Figure 6-21 Conveying Information via Graphics

Source: Author

Once again, similar to many of the things looked at in this chapter, not as much time has been put into the visuals of the current build as I would have liked. This is not to say that no thought or effort went into them, but merely to point out that there is a great deal of room for improvement in this area. *Unity* as a game engine, can support extremely visually appealing games if done correctly and any future iteration should take full advantage of that fact.

6.13 Audio

The current build of the game has no audio and it has not been incorporated in any stage of the development process. This game is, currently, completely silent, like most design software, which is to its determent. Anyone who has ever played a game knows how much even subtle audio ques can completely transform a player's experience of a game. Audio in game takes many forms, form a soundtrack that plays in the background of specific scenes or at specific parts of a game playthrough. Audio can also be something as simple as a simple sound being played when a particular action occurs, like when a puzzle piece is placed within a scene or when a player clicks on a certain button. The sound design in a game is hugely important and can help draw a player into a video game.

6.14 Conclusion

For this dissertation an educational 3D puzzle video game comprised of a *library of precedent puzzles* based off important works of architecture is proposed as a means of addressing one, the counter intuitive process of using shape grammar in digital applications where rules are design first before the making of the design as opposed to developing concurrently alongside the creation of the design through analysis, and two, the high barrier of entry for using shape grammar in digital design limiting its potential use for educational purposes with the goal of conveying tacit knowledge of architectural design.[3]

This dissertation does not seek to develop a fully developed and functional game ready to be published, rather it outlines the overall idea for this video game and looks at the development of an early protoype to test key gameplay mechanics. Chapter 1: Introduction looks at the research that has been done relating to shape grammar and outlines the issues this dissertation project seeks to tackle. This chapter also covers some basic information about game development and the termanology used throughout this document. How this idea of a 3D puzzle game came about as a potential solution the the problems stated before is outlined in Chapter 2: Scenarios. Chapter 3: Abstraction Methodology documents the generation of the Martin House precedent puzzle pieces to be used for this early prototype. Chapter 4: Gameplay outlines the core gameplay mechanics being looked at in this project while Chapter 5: Implementation details how those gameplay features were implemented in addition to how the puzzle levels iteslf was constructed in the *Unity*

editor. And finally, Chapter 6: Discussion & Framework for Future Development outlines where I would like to see the game's development go to from here.

The video game proposed by this dissertation is still a long way off from being completed and there is still much work to do, however, the first steps along this path have been taken.

---

[1] "What game narrative is and what it means in casual games," Medium Gaming, updated September 13, 2018, accessed March 3, 2019, https://medium.com/@alexstargame/what-game-narrative-is-and-what-it-means-in-casual-games-67f35c191424.

[2] Conor Linehan et al., "Learning curves: analysing pace and challenge in four successful puzzle games," (2014).

[3] Grasl and Economou, "From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter."; Park and Vakaló, "A Form-making Algorithm. Shape Grammar Reversed."; Piazzalunga and Fitzhorn, "Note on a three-dimensional shape grammar interpreter."; Sandstrom and Park, "Short Reflection in Action: An Educational Indie Video Game with Design Schema."; Tapia, "A visual implementation of a shape grammar system."; Tepavčević and Stojaković, "Shape grammar in contemporary architectural theory and design."; Trescak, Esteva, and Rodriguez, "A shape grammar interpreter for rectilinear forms."

Appendix A Diagram Keys

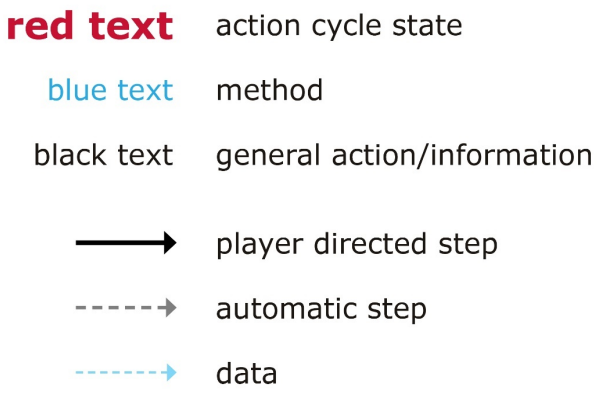


Figure 6-22 Flowchart Key

Source: Author

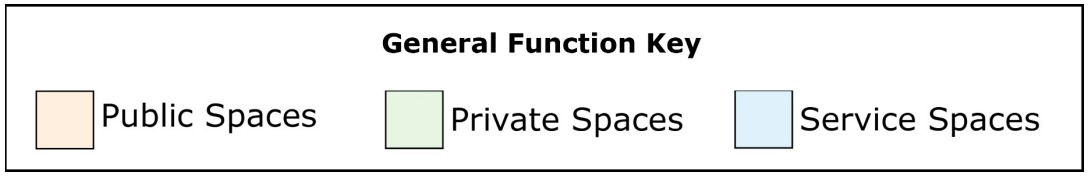


Figure 6-23 General Function Key

Source: Author

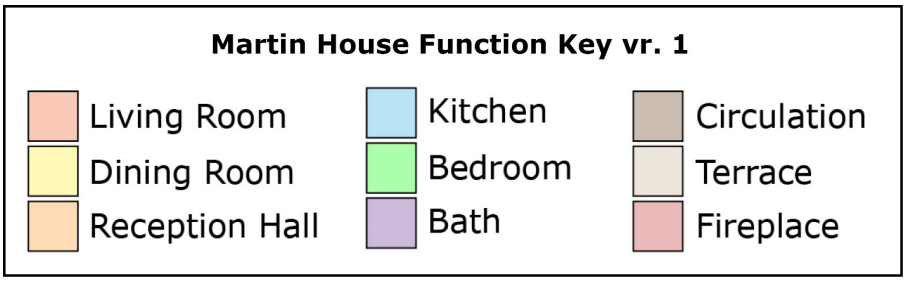


Figure 6-24 Martin House Abstraction Key for Iterations 1-2

Source: Author

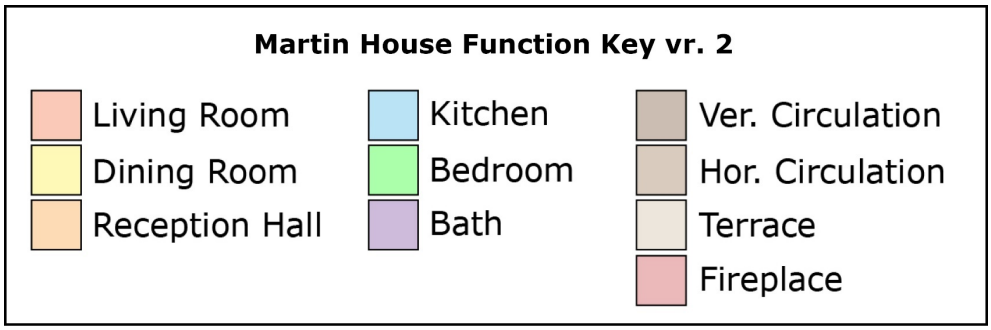


Figure 6-25 Martin House Abstraction Key for Iterations 3-4

Source: Author

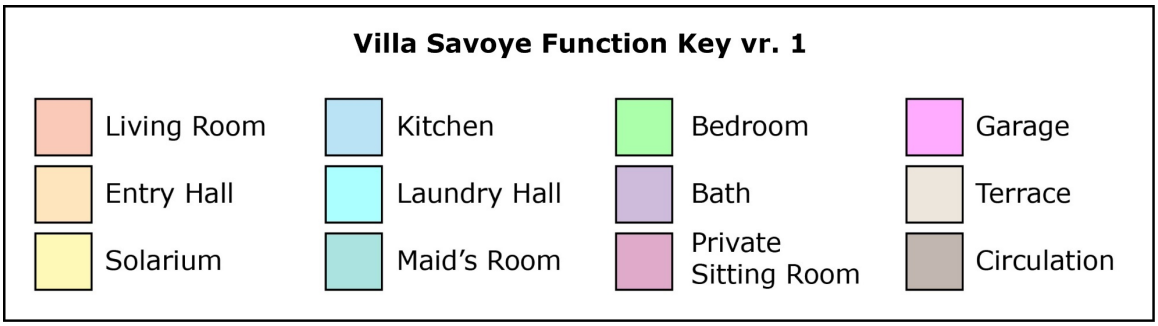


Figure 6-26 Villa Savoye Abstraction Key for Iterations 1-2

Source: Author

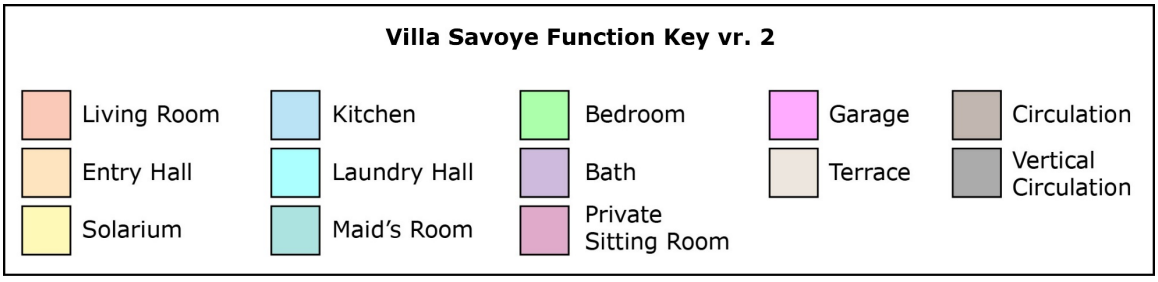


Figure 6-27 Villa Savoye Abstraction Key for Iterations 3-4

Source: Author

Appendix B Martin House Puzzle Pieces

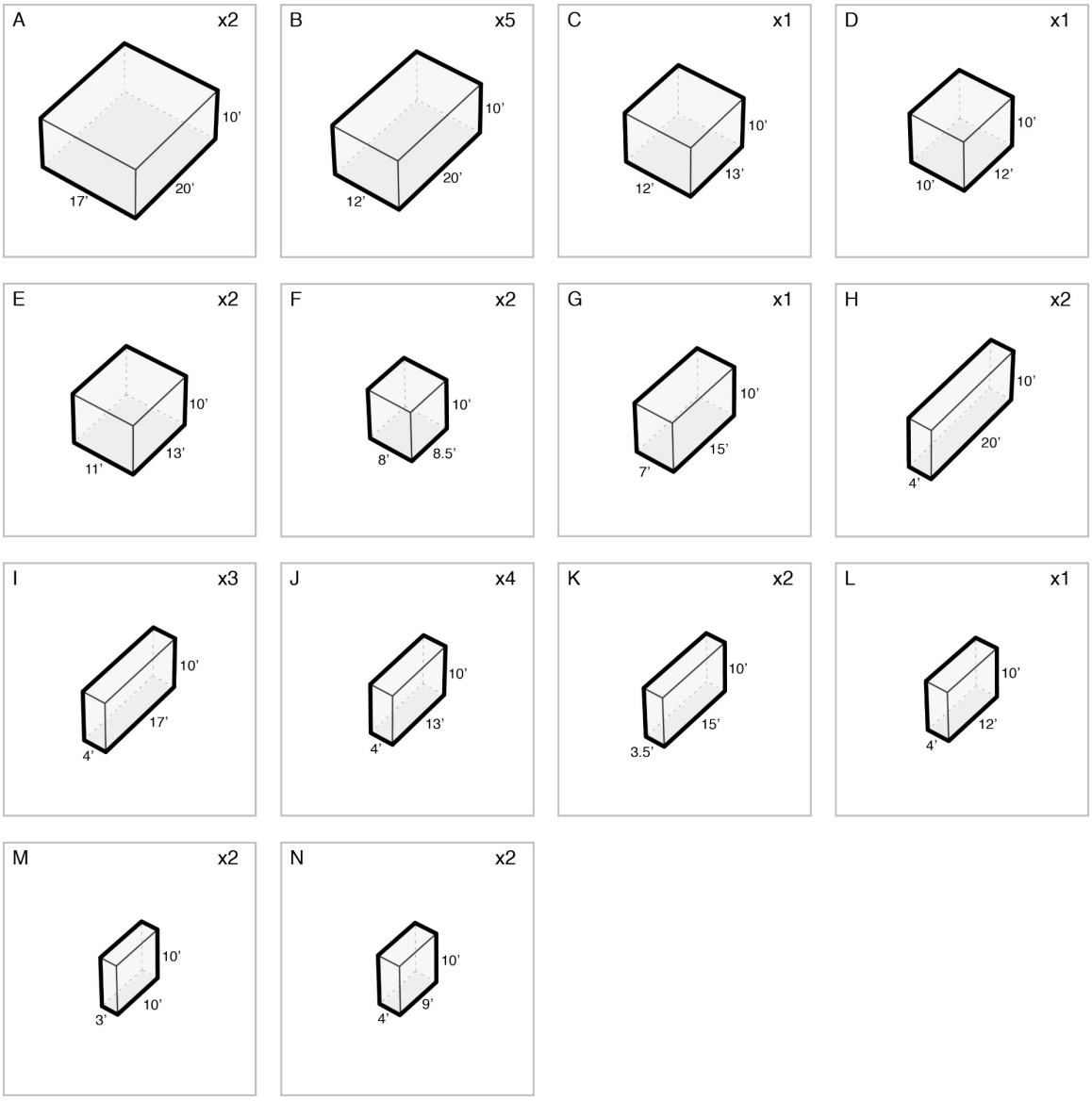


Figure 6-28 Martin House Iteration 4 – Basic Shapes

Source: Author

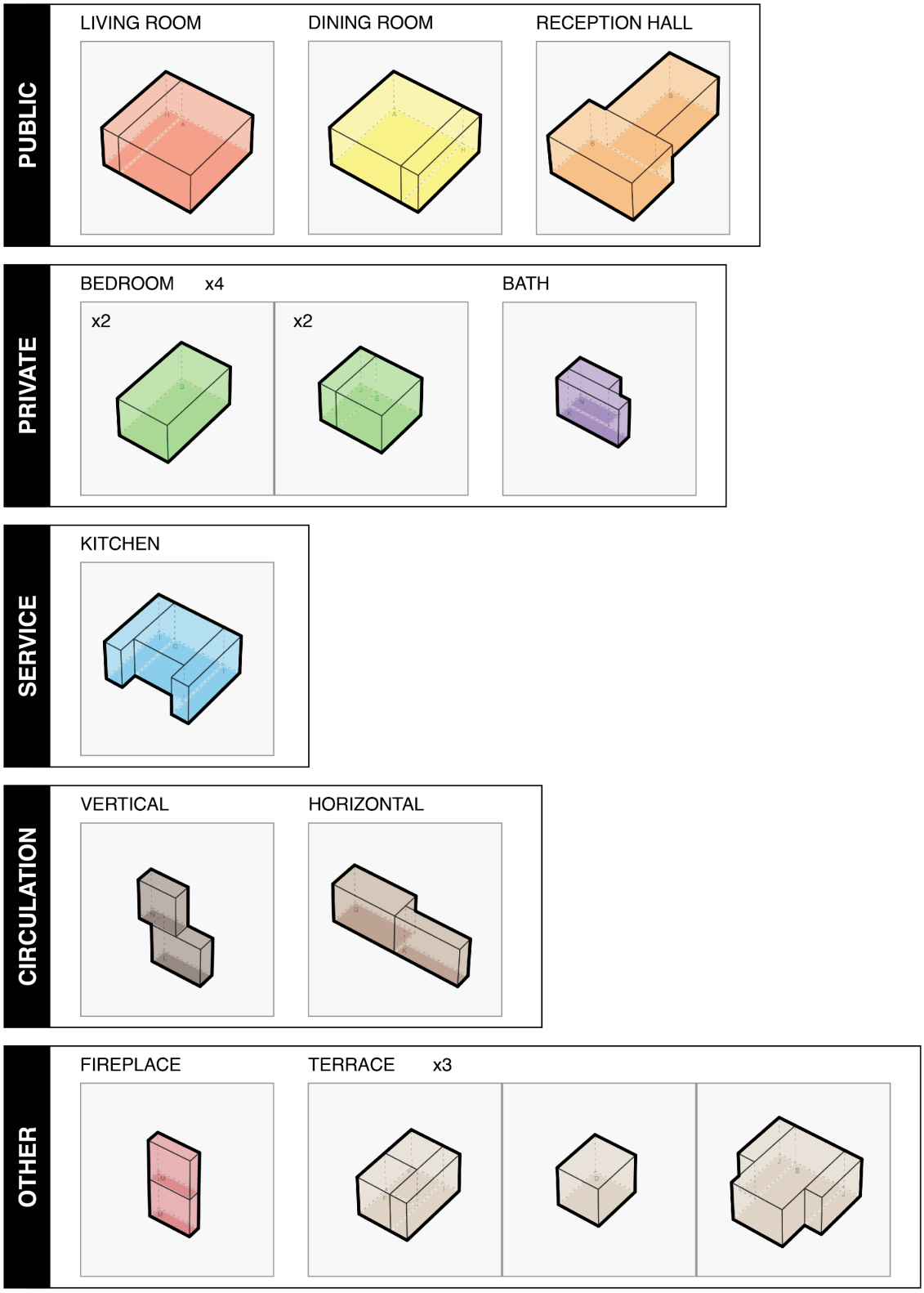


Figure 6-29 Martin House Iteration 4 – Labeled Shapes Provided to Player

Source: Author

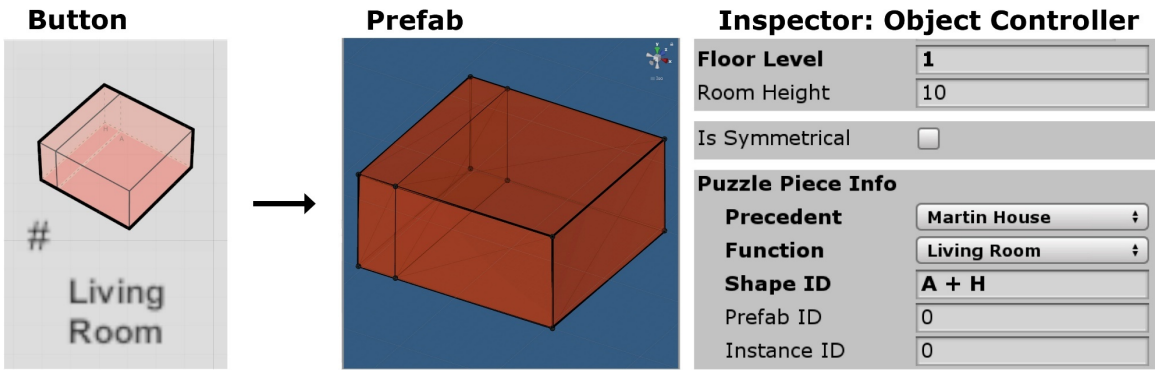


Figure 6-30 Living Room Button

Source: Author

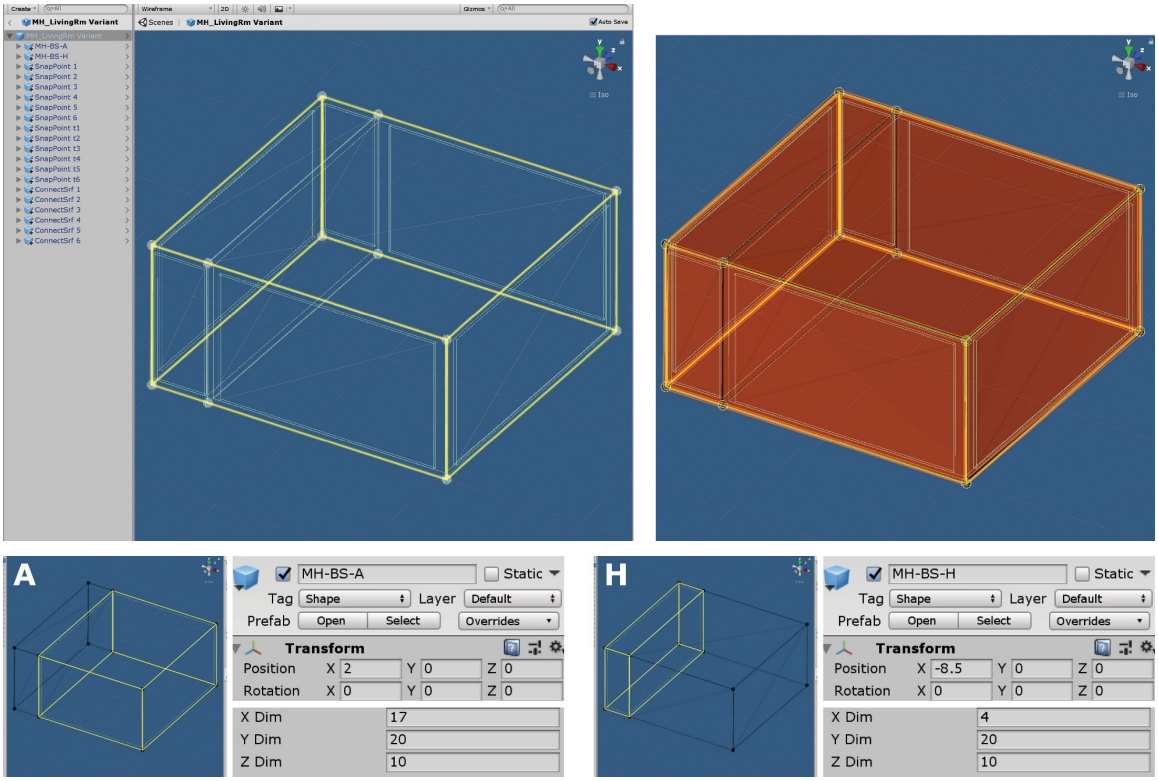


Figure 6-31 Living Room Shape Composition

Source: Author

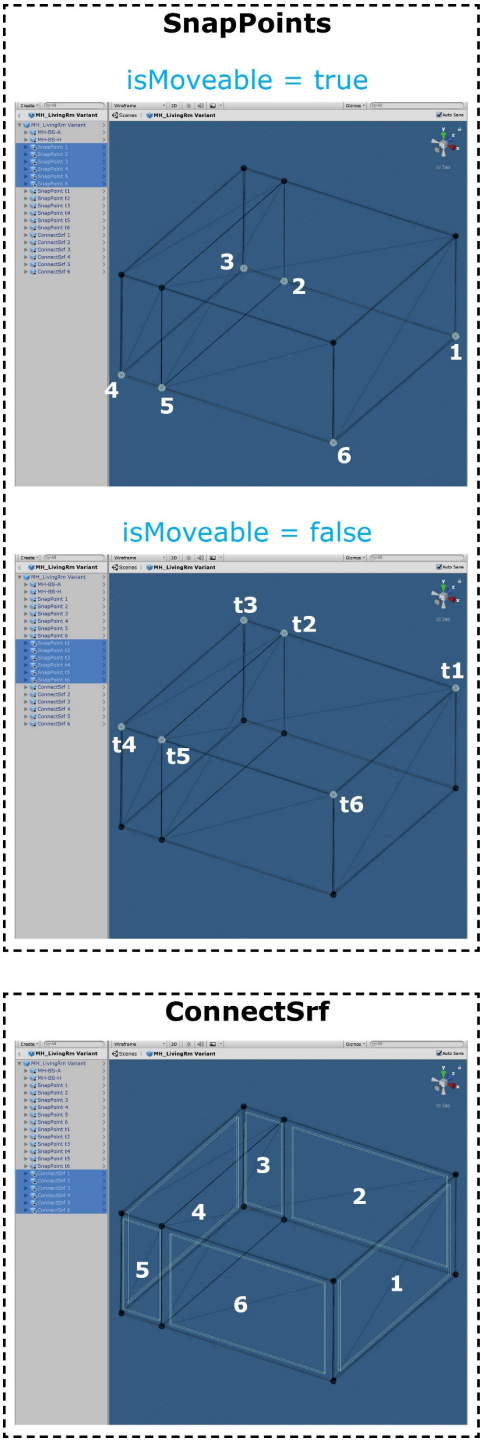


Figure 6-32 Living Room SnapPoint & ConnectSrf Locations

Source: Author

Table 6-1 Living Room Snapping Point Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	10.5	0	10	**t1**	10.5	10	10
2	-6.5	0	10	**t2**	-6.5	10	10
3	-10.5	0	10	**t3**	-10.5	10	10
4	-10.5	0	-10	**t4**	-10.5	10	-10
5	-6.5	0	-10	**t5**	-6.5	10	-10
6	10.5	0	-10	**t6**	10.5	10	-10

Table 6-2 Living Room Connection Surface Local Location & Sizing Values

	Position			Length
	X	y	z	
1	10.5	5	0	20
2	2	5	10	17
3	-8.5	5	10	4
4	-10.5	5	0	20
5	-8.5	5	-10	4
6	2	5	-10	17

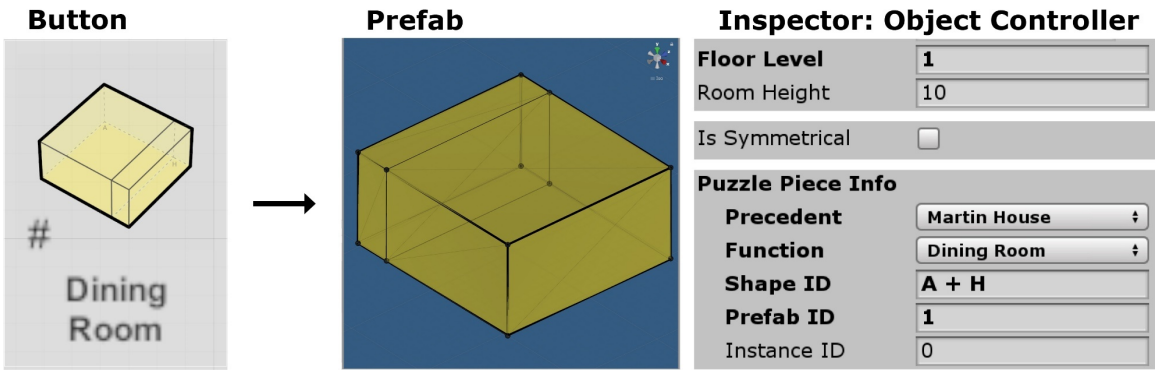


Figure 6-33 Dining Room Button

Source: Author

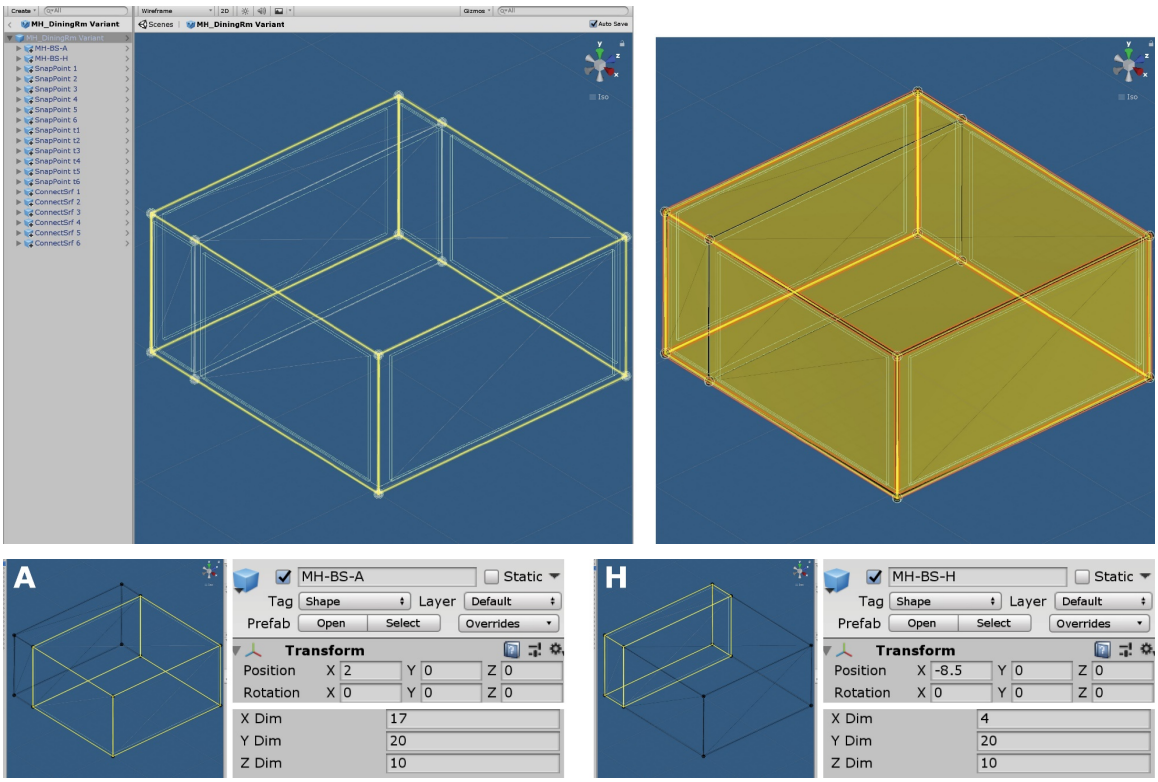


Figure 6-34 Dining Room Shape Composition

Source: Author

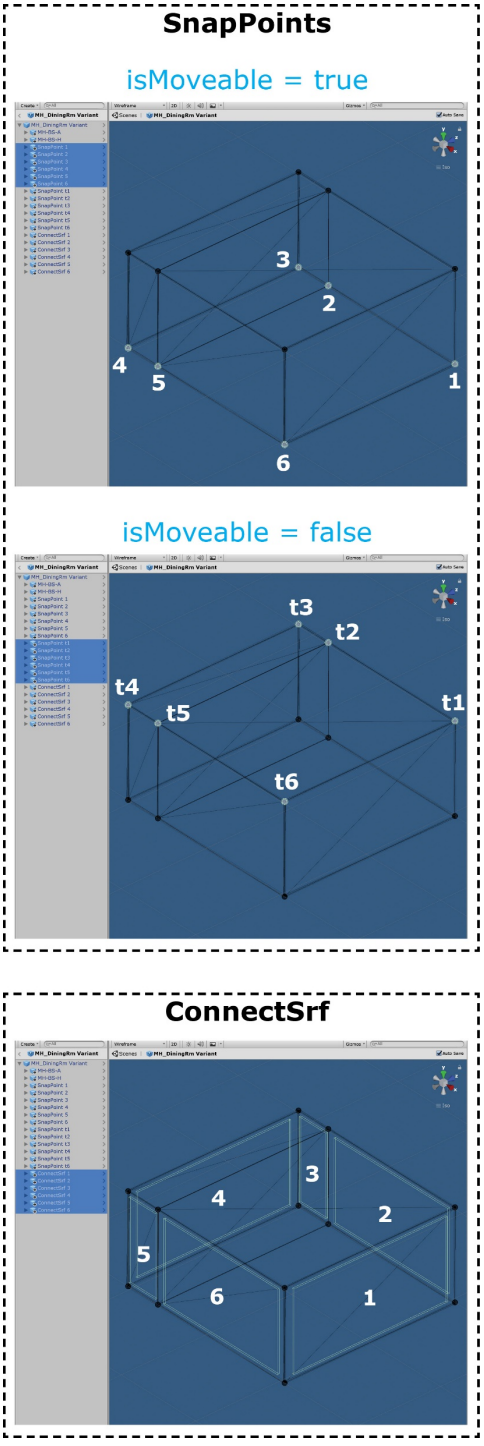


Figure 6-35 Dining Room SnapPoint & ConnectSrf Locations

Source: Author

Table 6-4 Dining Room Local Location Values

	Position		
	x	y	z
1	10.5	0	10
2	-6.5	0	10
3	-10.5	0	10
4	-10.5	0	-10
5	-6.5	0	-10
6	10.5	0	-10

	Position		
	x	y	z
t1	10.5	10	10
t2	-6.5	10	10
t3	-10.5	10	10
t4	-10.5	10	-10
t5	-6.5	10	-10
t6	10.5	10	-10

Table 6-3 Dining Room Connection Surface Local Location & Sizing Values

	Position			Length
	X	y	z	
1	10.5	5	0	20
2	2	5	10	17
3	-8.5	5	10	4
4	-10.5	5	0	20
5	-8.5	5	-10	4
6	2	5	-10	17

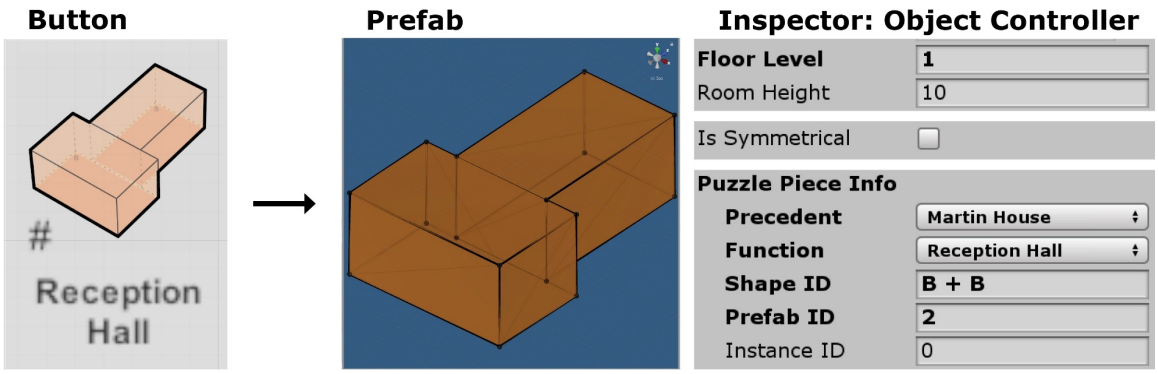


Figure 6-36 Reception Hall Button

Source: Author

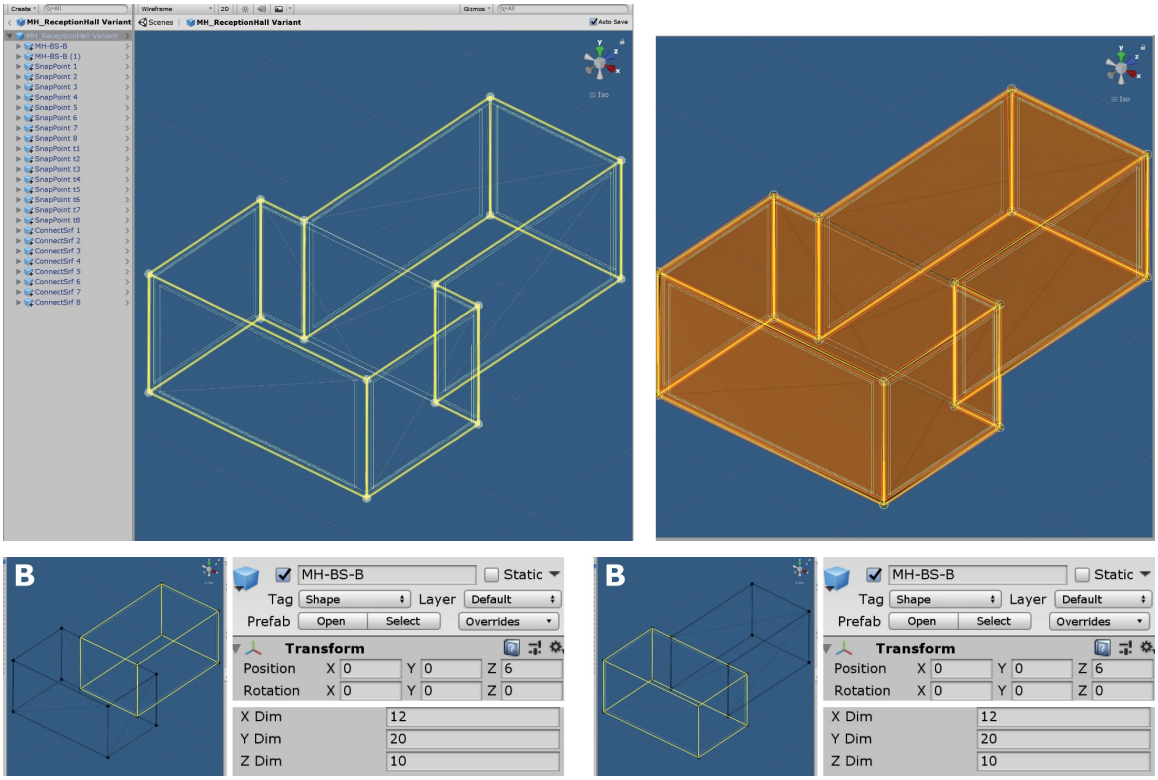


Figure 6-37 Reception Hall Shape Composition

Source: Author

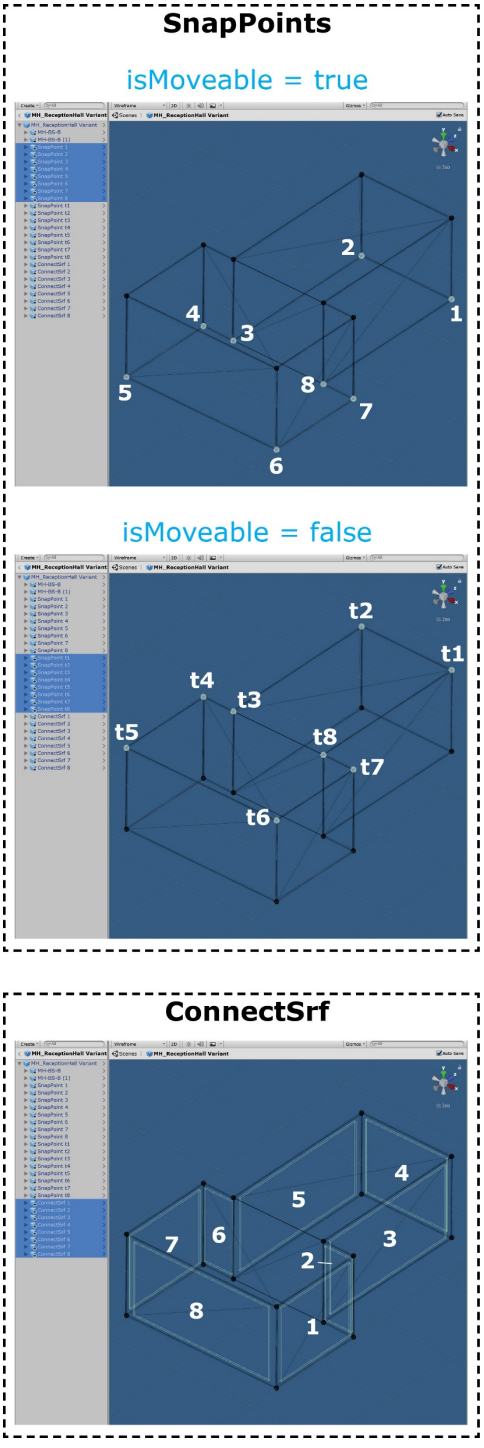


Figure 6-38 Reception Hall SnapPoint & ConnectSrf Locations

Source: Author

Table 6-6 Reception Hall Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	6	0	16	**t1**	6	10	16
2	-6	0	16	**t2**	-6	10	16
3	-6	0	-4	**t3**	-6	10	-4
4	-10	0	-4	**t4**	-10	10	-4
5	-10	0	-16	**t5**	-10	10	-16
6	10	0	-16	**t6**	10	10	-16
7	10	0	-4	**t7**	10	10	-4
8	6	0	-4	**t8**	6	10	-4

Table 6-5 Reception Hall Connection Surface Local Location & Sizing Values

	Position			Length
	X	y	z	
1	10	5	-10	12
2	8	5	-4	4
3	6	5	6	20
4	0	5	16	12
5	-6	5	6	20
6	-8	5	-4	4
7	-10	5	-10	12
8	0	5	-16	20

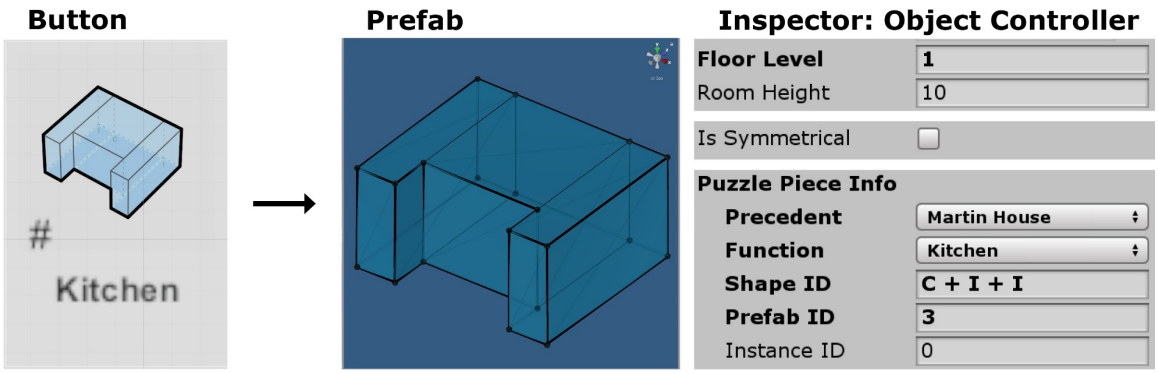


Figure 6-39 Kitchen Button

Source: Author

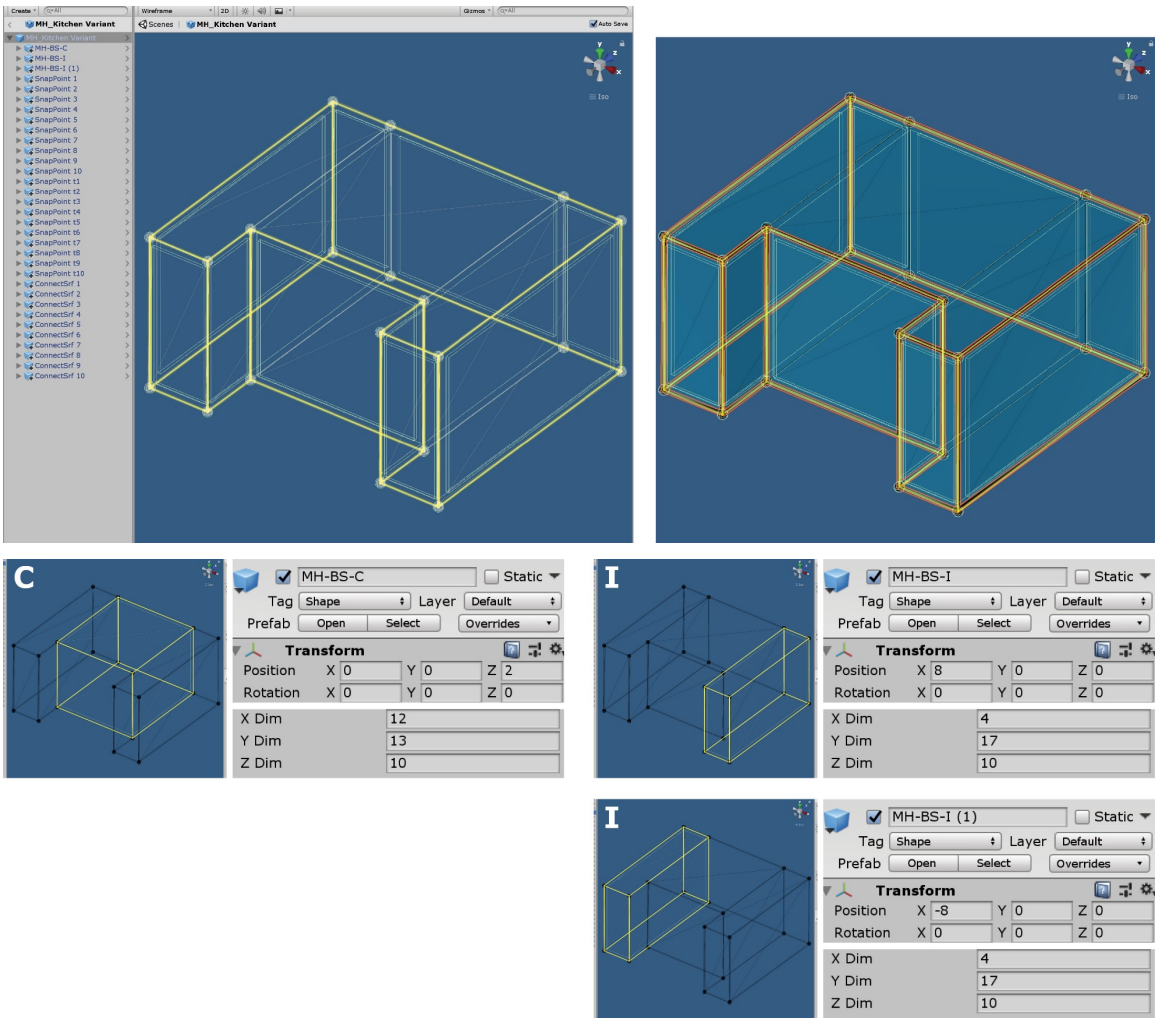


Figure 6-40 Kitchen Shape Composition

Source: Author

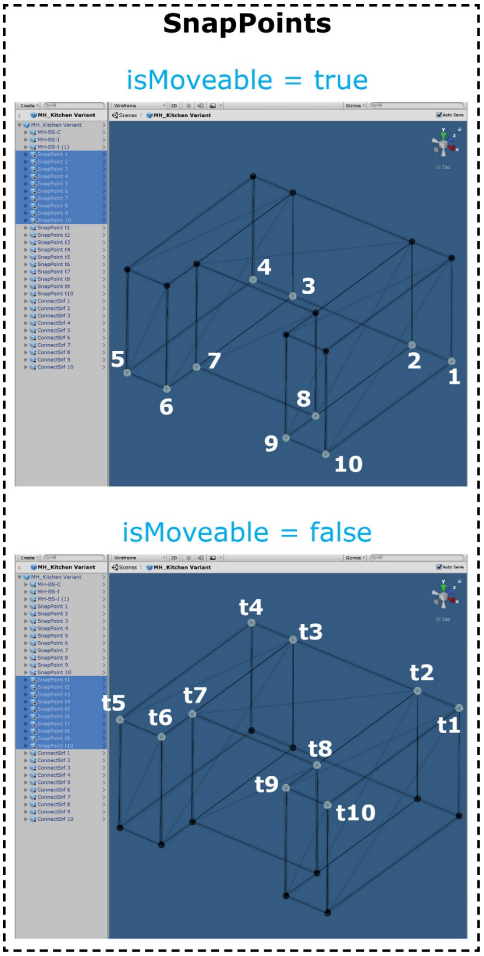


Figure 6-41 Kitchen Snapping SnapPoint & ConnectSrf Locations Source: Author

Table 6-7 Kitchen Snapping Point Local Location Values

	Position						
	x	y	z		x	y	z
1	10	0	8.5	**t1**	10	10	8.5
2	6	0	8.5	**t2**	6	10	8.5
3	-6	0	8.5	**t3**	-6	10	8.5
4	-10	0	8.5	**t4**	-10	10	8.5
5	-10	0	-8.5	**t5**	-10	10	-8.5
6	-6	0	-8.5	**t6**	-6	10	-8.5
7	-6	0	-4.5	**t7**	-6	10	-4.5
8	6	0	-4.5	**t8**	6	10	-4.5
9	6	0	-8.5	**t9**	6	10	-8.5
10	10	0	-8.5	**t10**	10	10	-8.5

Table 6-8 Kitchen Connection Surface Local Location and Sizing Values

	Position			Length
	X	y	z	
1	10	5	0	17
2	8	5	8.5	4
3	0	5	8.5	12
4	-8	5	8.5	4
5	-10	5	0	17
6	-8	5	-8.5	4
7	-6	5	-6.5	4
8	0	5	-4.5	12
9	6	5	-6.5	4
10	8	5	-8.5	4

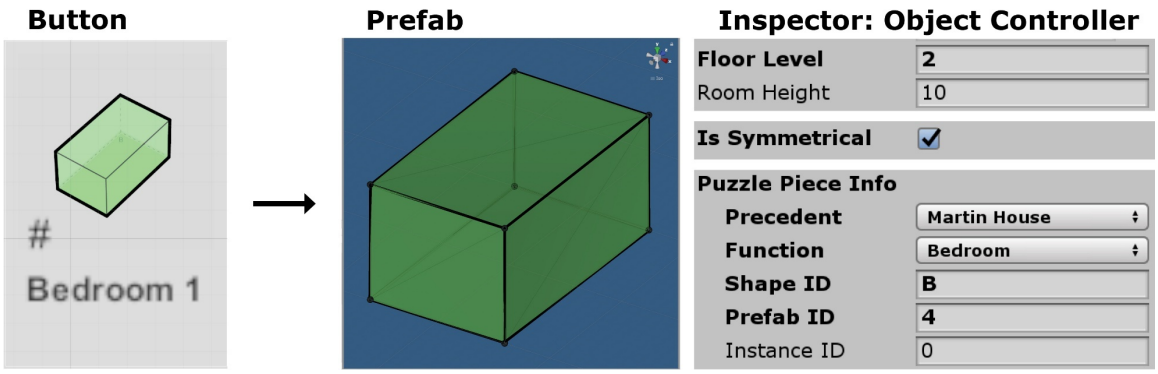


Figure 6-42 Bedroom 1 Button

Source: Author

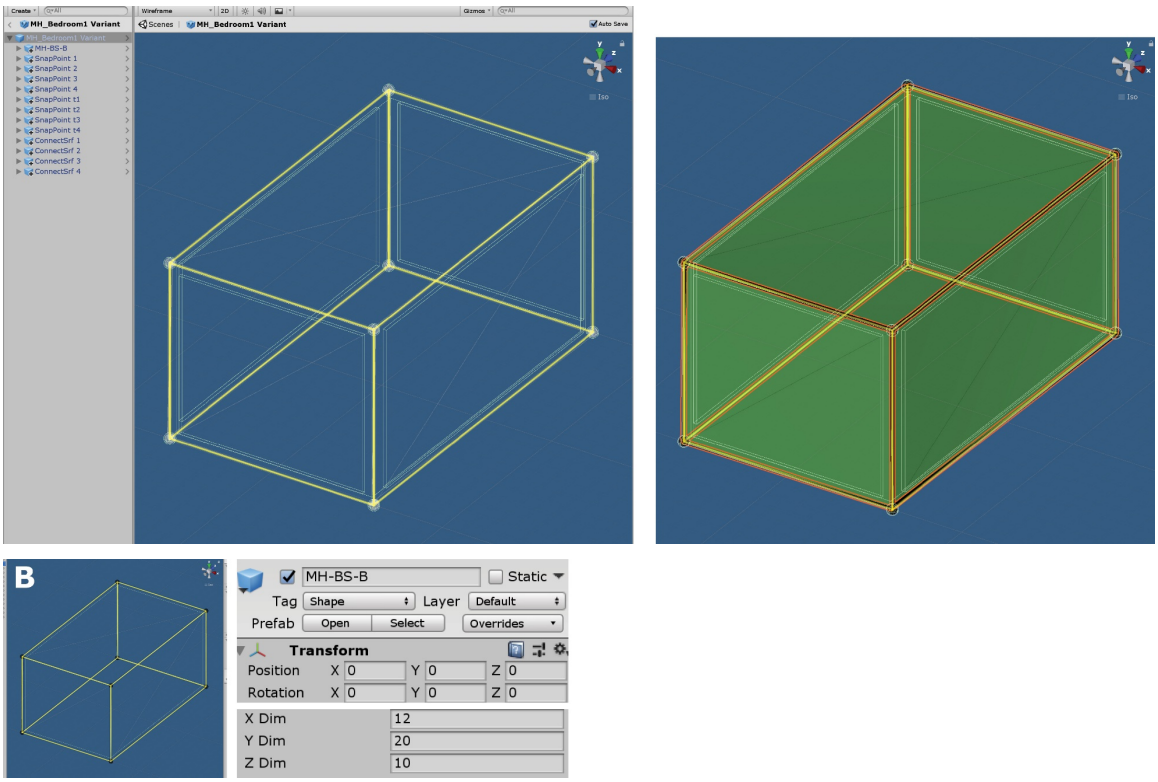


Figure 6-43 Bedroom 1 Shape Composition

Source: Author

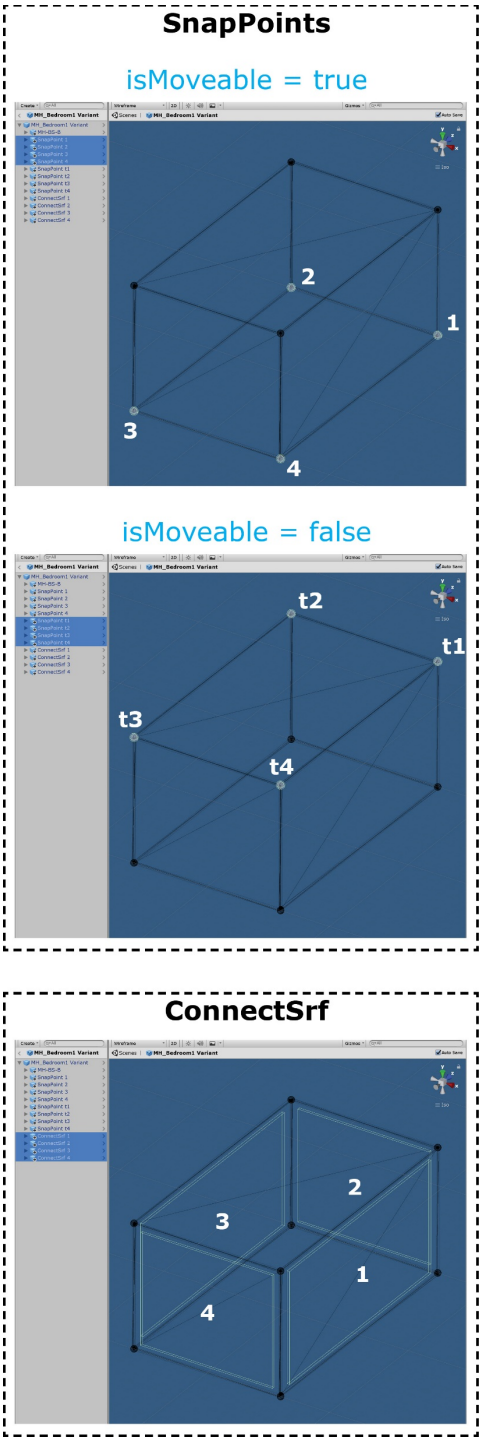


Figure 6-44 Bedroom 1 SnapPoint & ConnectSrf Locations
Source: Author

Table 6-9 Bedroom 1 SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	6	0	10	**t1**	6	10	10
2	-6	0	10	**t2**	-6	10	10
3	-6	0	-10	**t3**	-6	10	-10
4	6	0	-10	**t4**	6	10	-10

Table 6-10 Bedroom 1 ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	6	5	0	20
2	0	5	10	12
3	-6	5	0	20
4	0	5	-10	12

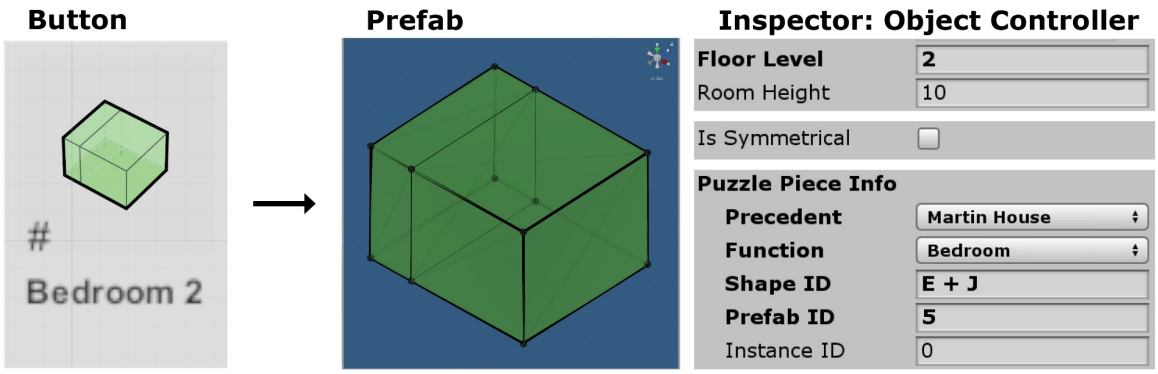


Figure 6-45 Bedroom 2 Button

Source: Author

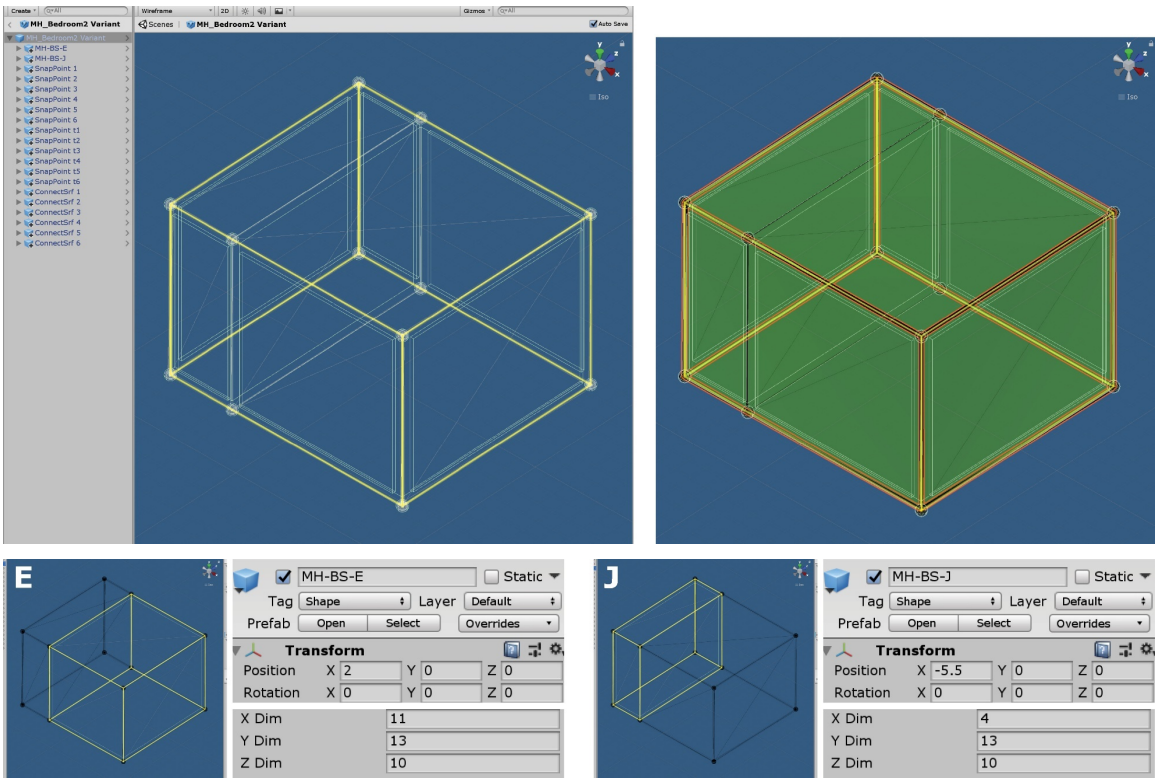


Figure 6-46 Bedroom 2 Shape Composition

Source: Author

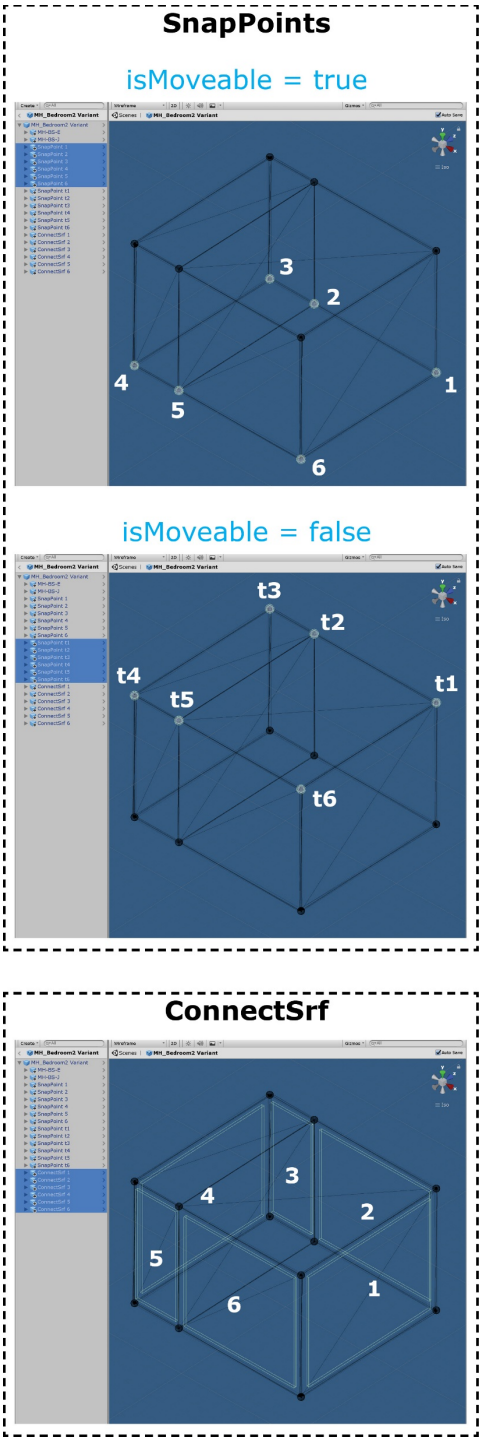


Figure 6-47 Bedroom 2 SnapPoint & ConnectSrf Locations
Source: Author

Table 6-11 Bedroom 2 SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	7.5	0	6.5	**t1**	7.5	10	6.5
2	-3.5	0	6.5	**t2**	-3.5	10	6.5
3	-7.3	0	6.5	**t3**	-7.3	10	6.5
4	-7.3	0	-6.5	**t4**	-7.3	10	-6.5
5	-3.5	0	-6.5	**t5**	-3.5	10	-6.5
6	7.3	0	-6.5	**t6**	7.3	10	-6.5

Table 6-12 Bedroom 2 ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	7.5	5	0	13
2	2	5	6.5	11
3	-5.5	5	6.5	4
4	-7.5	5	0	13
5	-5.5	5	-6.5	4
6	2	5	-6.5	11

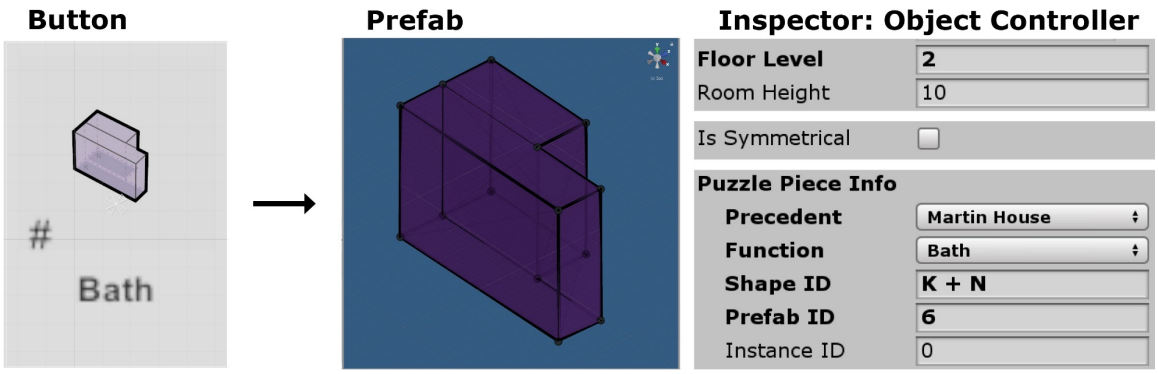


Figure 6-48 Bath Button

Source: Author

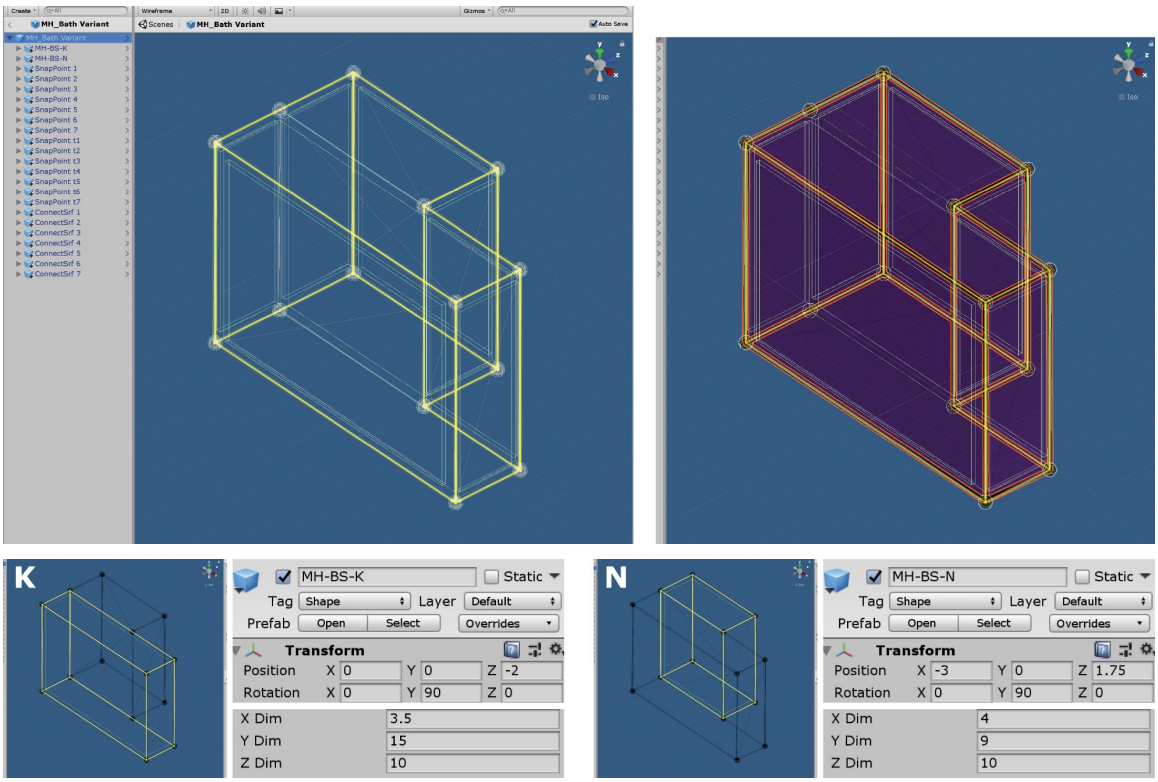


Figure 6-49 Bath Shape Composition

Source: Author

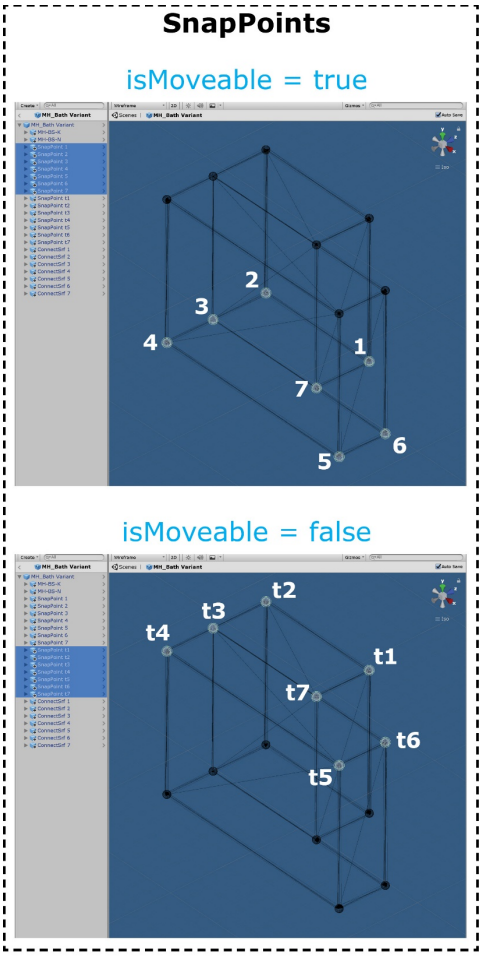


Figure 6-50 Bath SnapPoint & ConnectSrf Locations
Source: Author

Table 6-14 Bath SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	1.5	0	3.75	**t1**	1.5	10	3.75
2	-7.5	0	3.75	**t2**	-7.5	10	3.75
3	-7.5	0	-0.25	**t3**	-7.5	10	-0.25
4	-7.5	0	-3.75	**t4**	-7.5	10	-3.75
5	7.5	0	-3.75	**t5**	7.5	10	-3.75
6	7.5	0	-0.25	**t6**	7.5	10	-0.25
7	1.5	0	-0.25	**t7**	1.5	10	-0.25

Table 6-13 Bath ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	1.5	5	1.75	4
2	-3	5	3.75	9
3	-7.5	5	1.75	4
4	-7.5	5	-2	3.5
5	0	5	-3.75	15
6	7.5	5	-2	3.5
7	4.5	5	-0.25	6

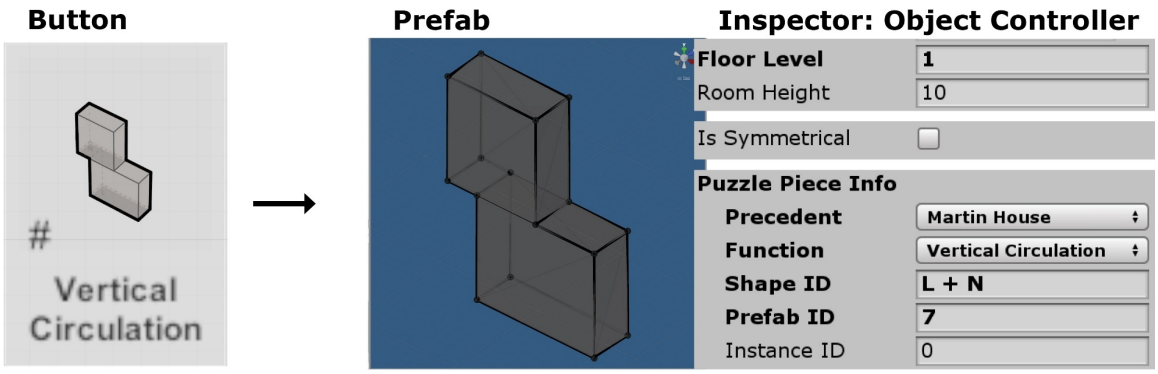


Figure 6-51 Vertical Circulation Button

Source: Author

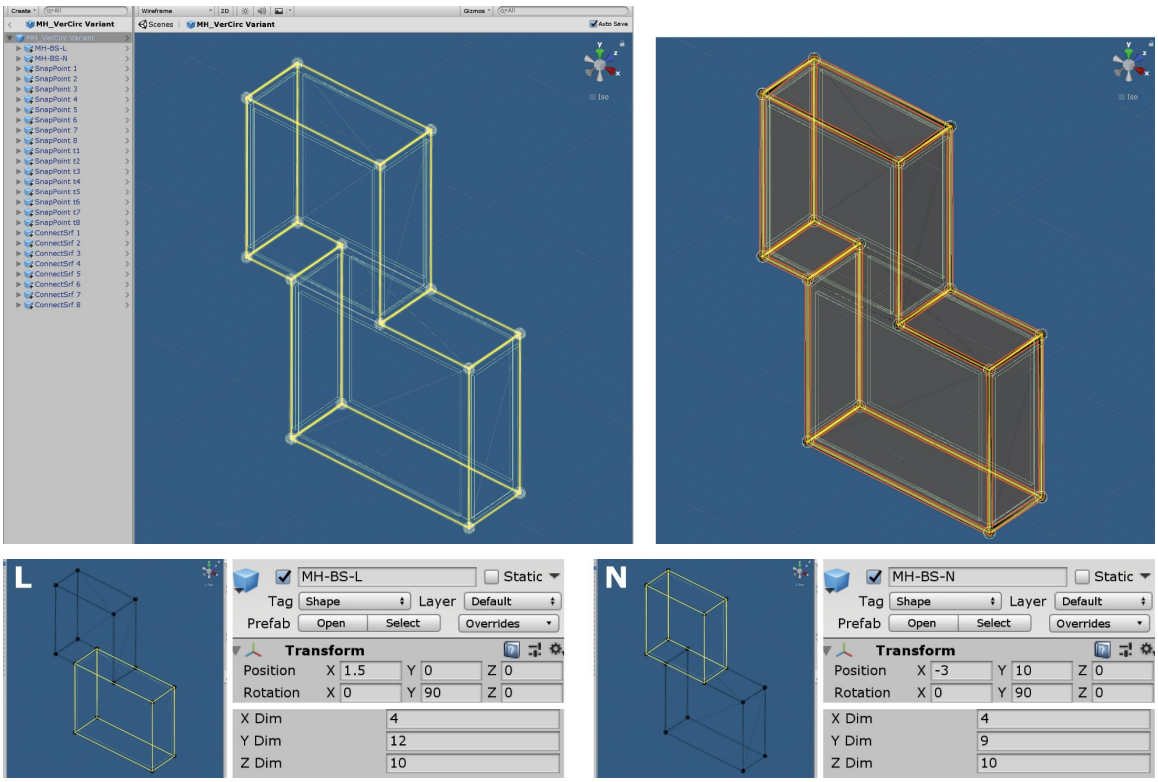


Figure 6-52 Vertical Circulation Shape Composition

Source: Author

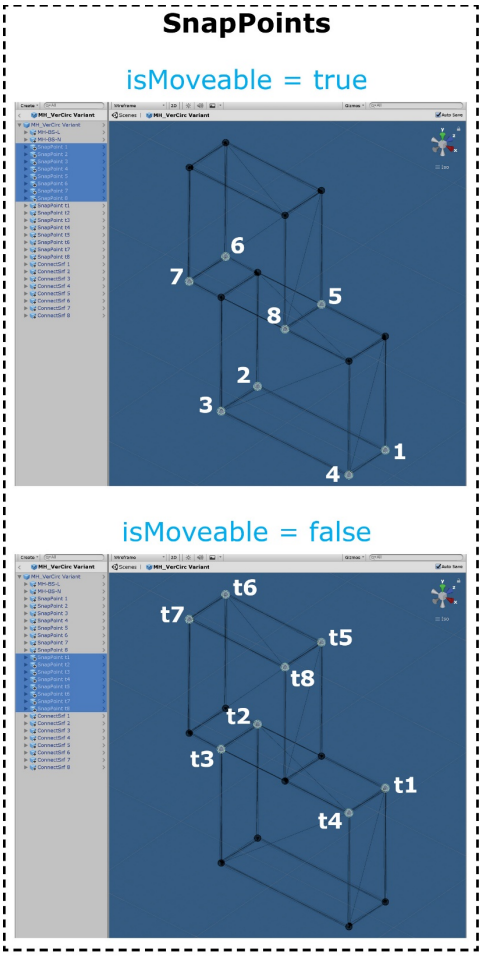


Figure 6-53 Vertical Circulation SnapPoint & ConnectSrf Locations
Source: Author

Table 6-15 Vertical Circulation SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	7.5	0	2	**t1**	7.5	10	2
2	-4.5	0	2	**t2**	-4.5	10	2
3	-4.5	0	-2	**t3**	-4.5	10	-2
4	7.5	0	-2	**t4**	7.5	10	-2
5	1.5	10	2	**t5**	1.5	20	2
6	-7.5	10	2	**t6**	-7.5	20	2
7	-7.5	10	-2	**t7**	-7.5	20	-2
8	1.8	10	-2	**t8**	1.8	20	-2

Table 6-16 Vertical Circulation ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	7.6	5	0	4
2	1.5	5	2	12
3	-4.5	5	0	4
4	1.5	5	-2	12
5	-1.5	15	0	4
6	-3	15	2	9
7	-7.5	15	0	4
8	-3	15	-2	9

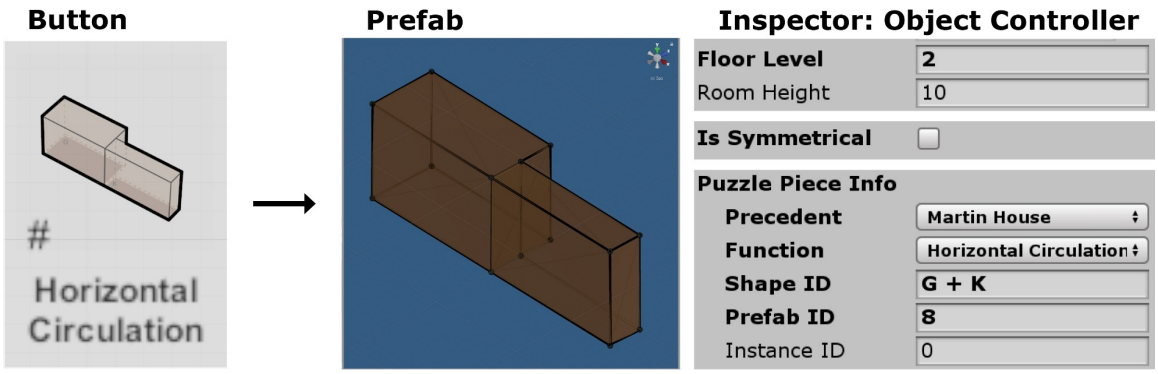


Figure 6-54 Horizontal Circulation Button

Source: Author

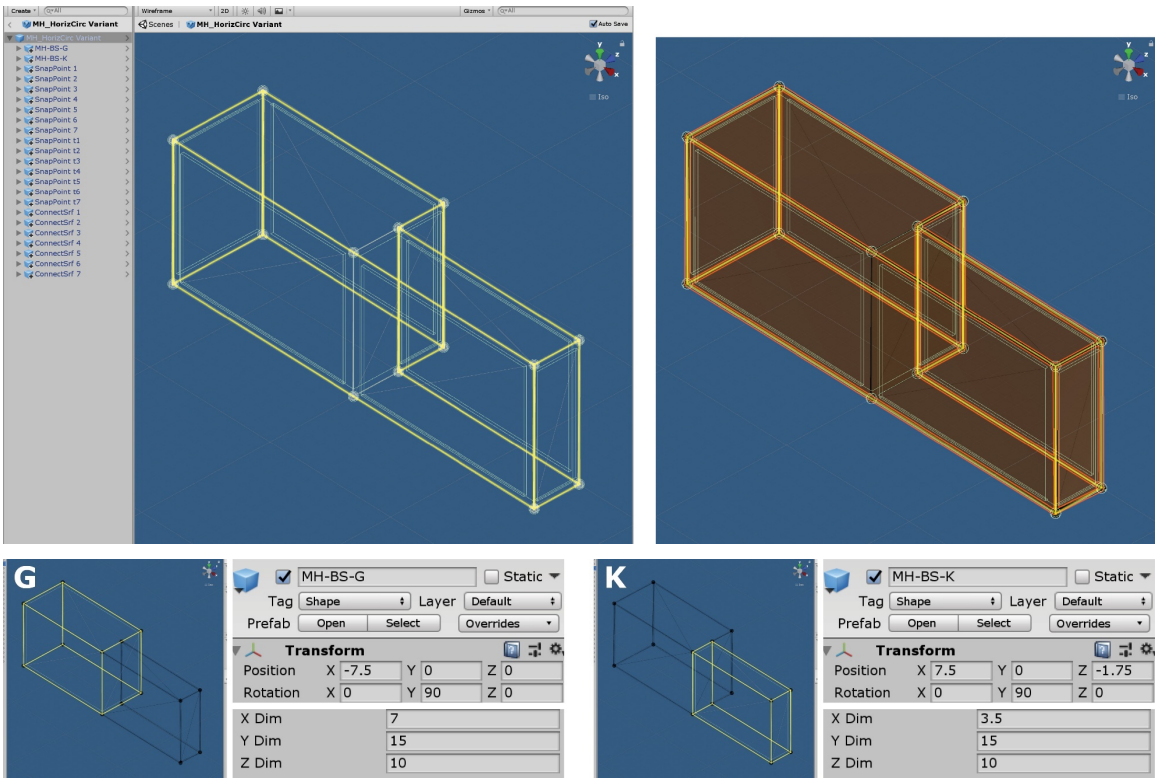


Figure 6-55 Horizontal Circulation Shape Composition

Source: Author

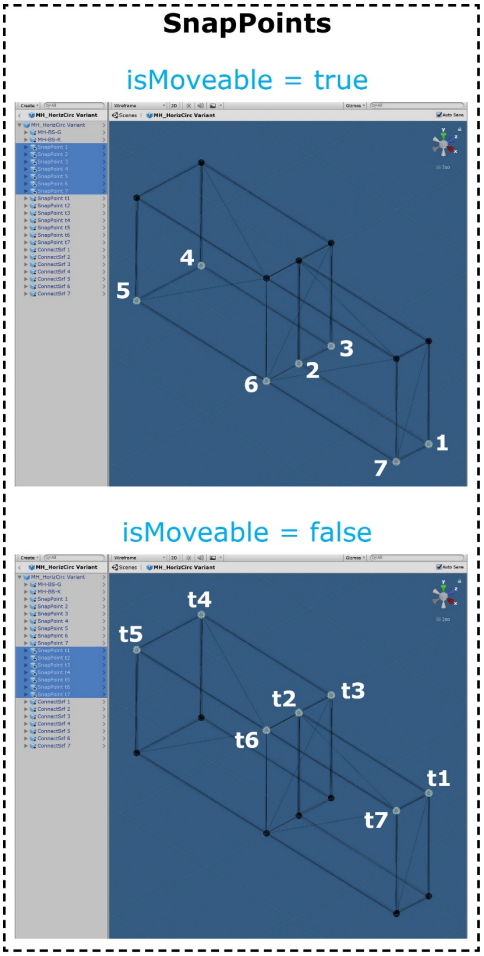


Figure 6-56 Horizontal Circulation SnapPoint & ConnectSrf Locations Source: Author

Table 6-17 Horizontal Circulation SnapPoint' Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	15	0	0	**t1**	15	10	0
2	0	0	0	**t2**	0	10	0
3	0	0	3.5	**t3**	0	10	3.5
4	-15	0	3.5	**t4**	-15	10	3.5
5	-15	0	-3.5	**t5**	-15	10	-3.5
6	0	0	-3.5	**t6**	0	10	-3.5
7	15	0	-3.5	**t7**	15	10	-3.5

Table 6-18 Horizontal Circulation ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	0	5	1.75	3.5
2	-7.5	5	3.5	15
3	-15	5	0	7
4	-7.5	5	-3.5	15
5	7.5	5	-3.5	15
6	15	5	-1.75	3.5
7	7.5	5	0	15

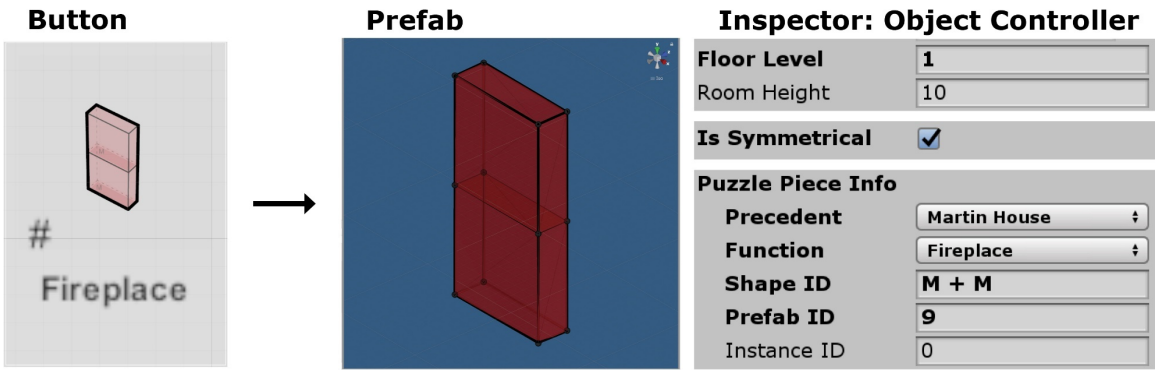


Figure 6-57 Fireplace Button

Source: Author

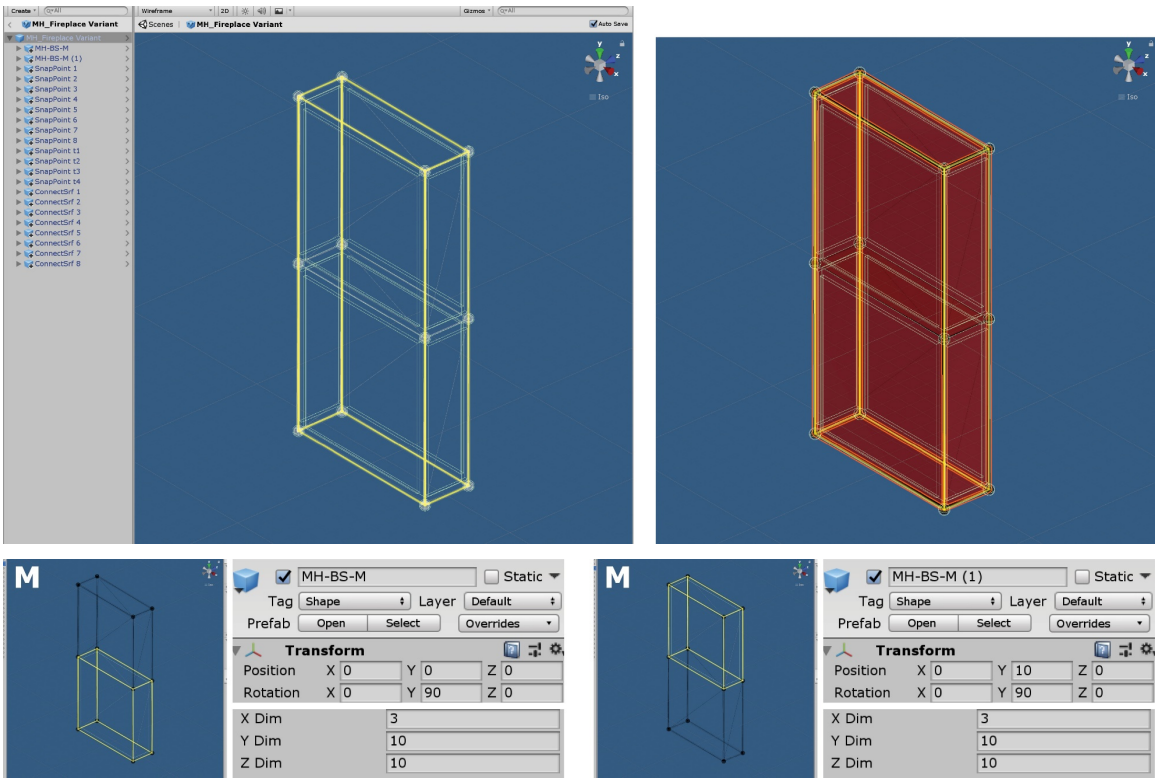


Figure 6-58 Fireplace Shape Composition

Source: Author

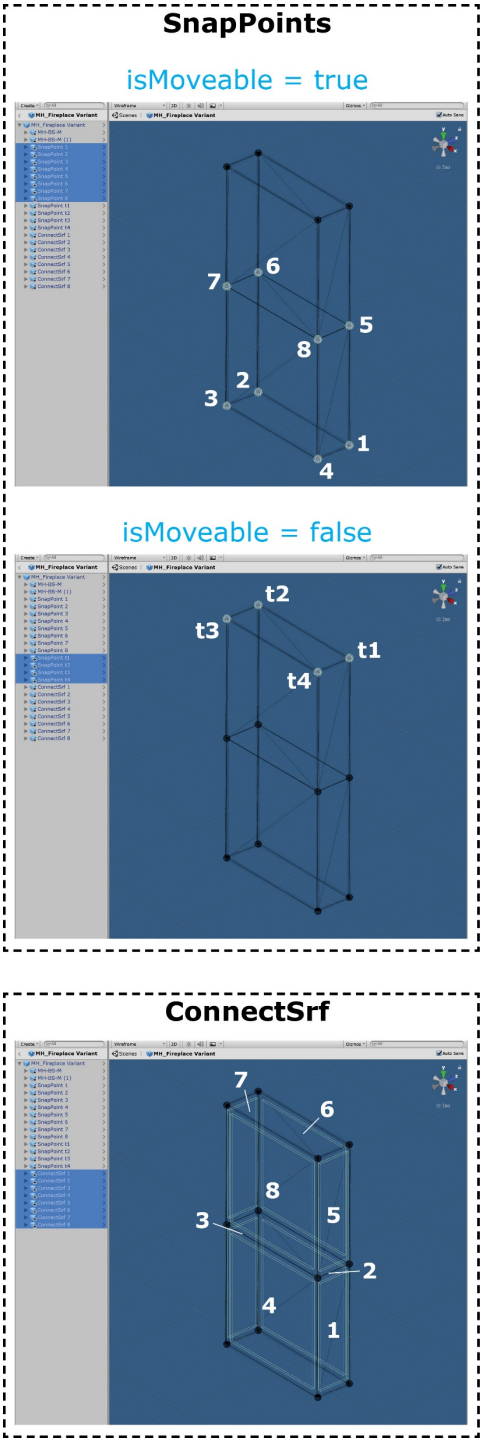


Figure 6-59 Fireplace Snapping SnapPoint & ConnectSrf Locations Source: Author

Table 6-19 Fireplace's SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	5	0	1.5	**t1**	5	10	1.5
2	-5	0	1.5	**t2**	-5	10	1.5
3	-5	0	-1.5	**t3**	-5	10	-1.5
4	5	0	-1.5	**t4**	5	10	-1.5
5	5	10	1.5	**t5**	5	20	1.5
6	-5	10	1.5	**t6**	-5	20	1.5
7	-5	10	-1.5	**t7**	-5	20	-1.5
8	5	10	-1.5	**t8**	5	20	-1.5

Table 6-20 Fireplace's ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	5	5	0	3
2	0	5	1.5	10
3	-5	5	0	3
4	0	5	-1.5	10
5	5	15	0	3
6	0	15	1.5	10
7	-5	15	0	3
8	0	15	-1.5	10

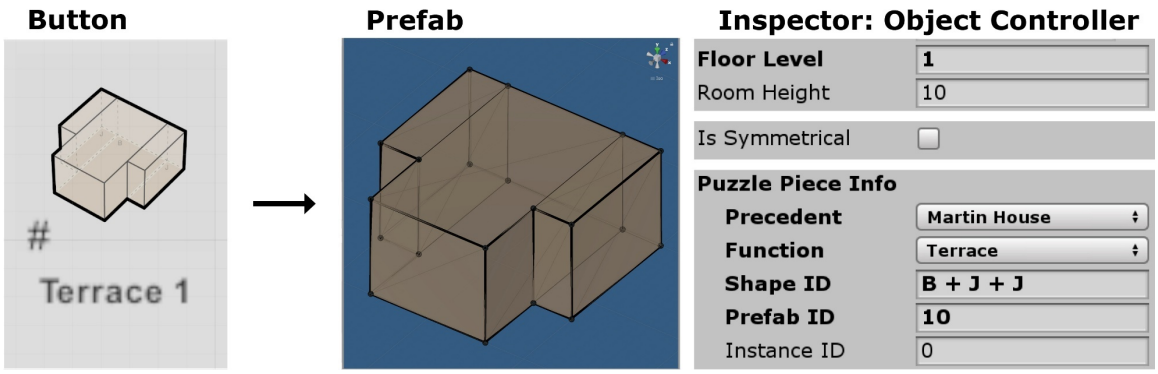


Figure 6-60 Terrace 1 Button

Source: Author

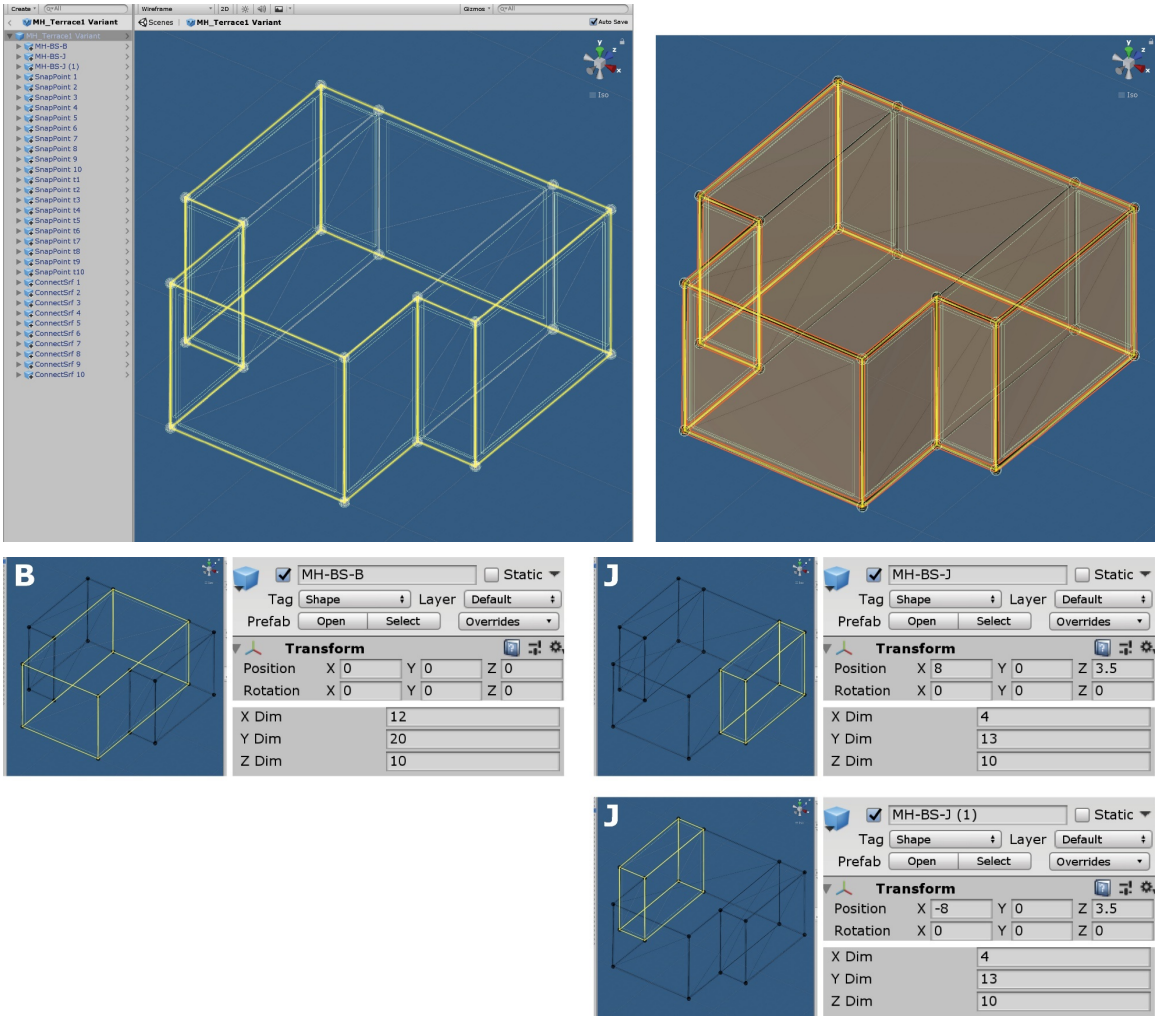


Figure 6-61 Terrace 1 Shape Composition

Source: Author

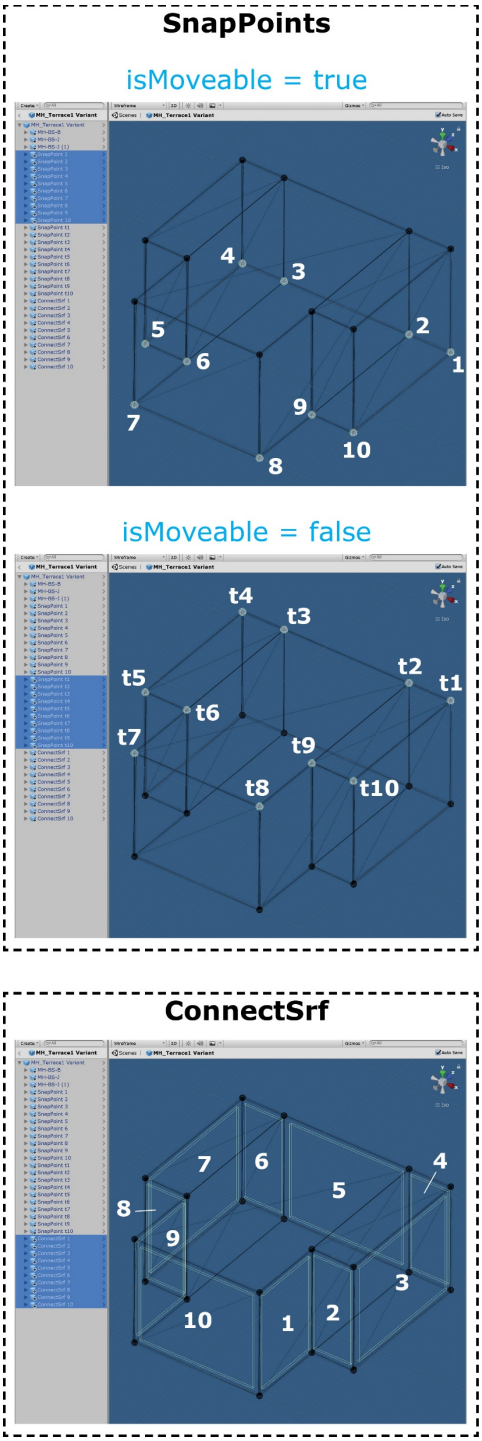


Figure 6-62 Terrace 1's SnapPoint & ConnectSrf Locations

Source: Author

Table 6-21 Terrace 1's SnapPoint Local Location & Sizing Values

	Position				Position		
	x	y	z		x	y	z
1	10	0	10	**t1**	10	10	10
2	6	0	10	**t2**	6	10	10
3	-6	0	10	**t3**	-6	10	10
4	-10	0	10	**t4**	-10	10	10
5	-10	0	-3	**t5**	-10	10	-3
6	-6	0	-3	**t6**	-6	10	-3
7	-6	0	-10	**t7**	-6	10	-10
8	6	0	-10	**t8**	6	10	-10
9	6	0	-3	**t9**	6	10	-3
10	10	0	-3	**t10**	10	10	-3

Table 6-22 Terrace 1's ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	6	5	-6.5	7
2	8	5	-3	4
3	10	5	3.5	13
4	8	5	10	4
5	0	5	10	12
6	-8	5	10	4
7	-10	5	3.5	13
8	-8	5	-3	4
9	-6	5	-6.6	7
10	0	5	-10	12

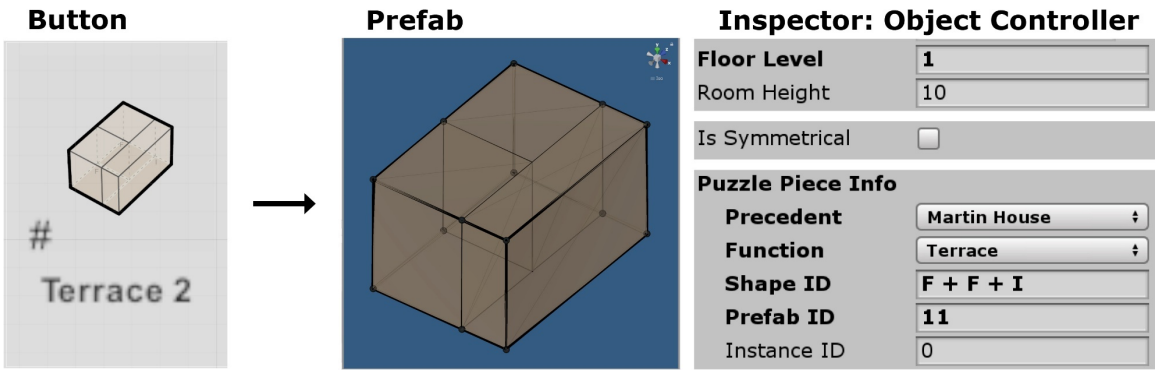


Figure 6-63 Terrace 2 Button

Source: Author

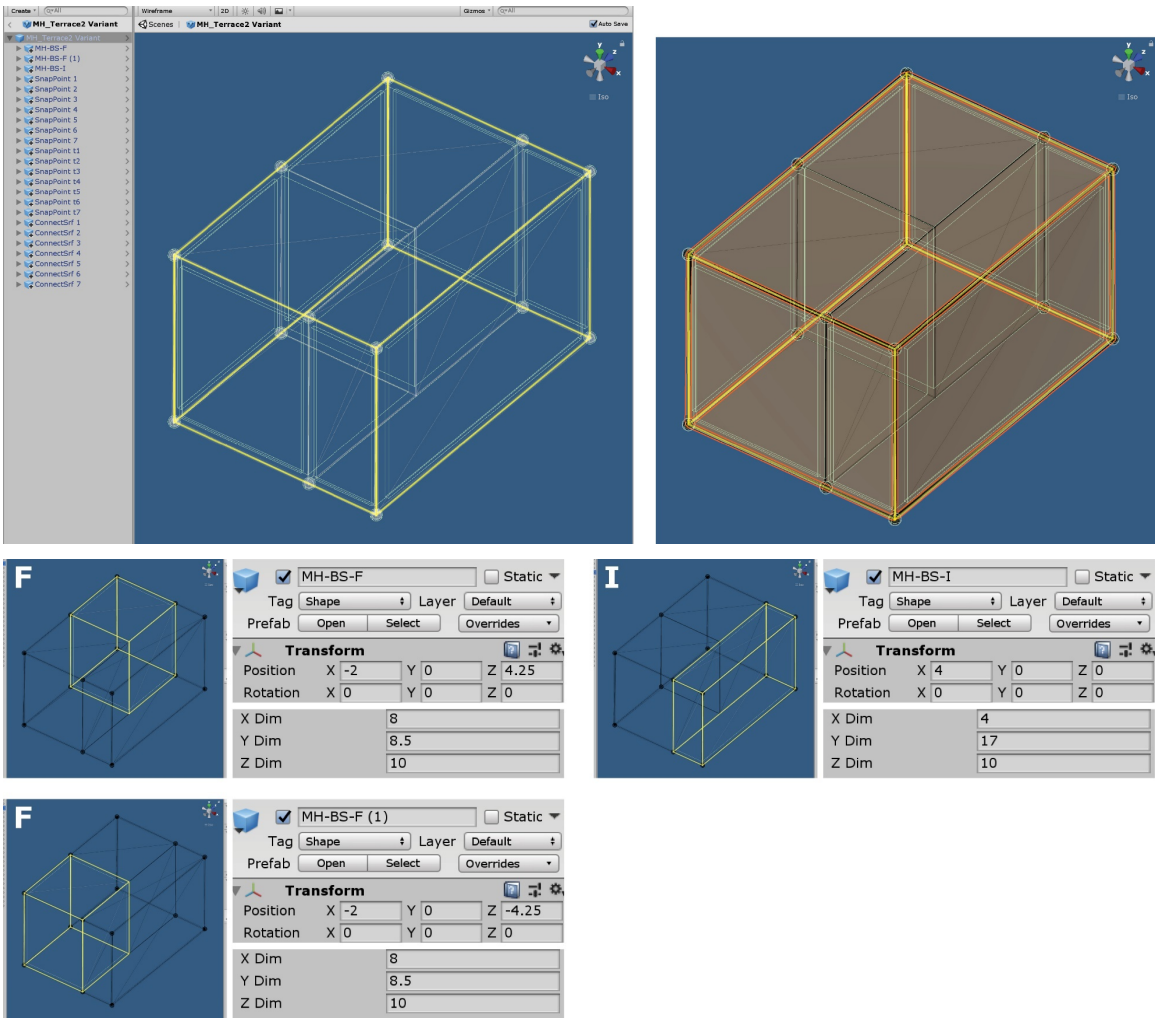


Figure 6-64 Terrace 2 Shape Composition

Source: Author

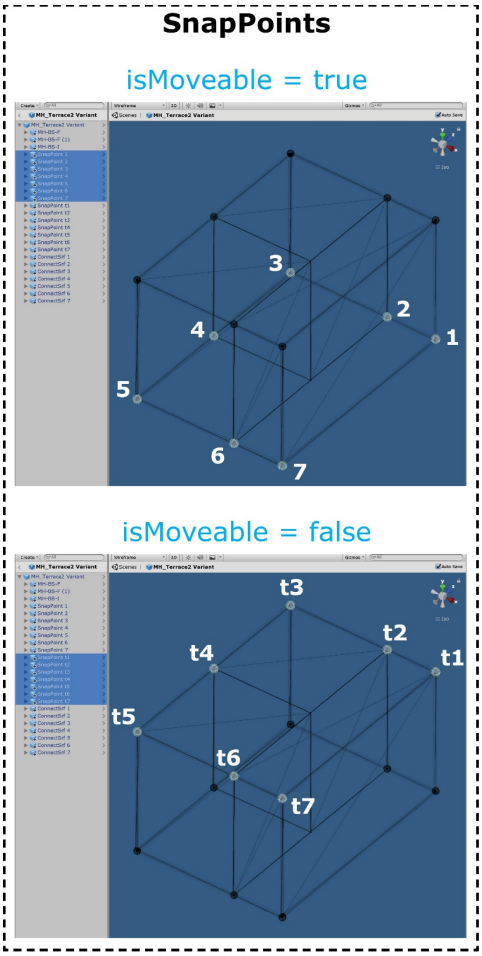


Figure 6-65 Terrace 2's SnapPoint & ConnectSrf Locations

Source: Author

Table 6-23 Terrace 2's SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	6	0	8.5	**t1**	6	10	8.5
2	2	0	8.5	**t2**	2	10	8.5
3	-6	0	8.5	**t3**	-6	10	8.5
4	-6	0	0	**t4**	-6	10	0
5	-6	0	-8.5	**t5**	-6	10	-8.5
6	2	0	-8.5	**t6**	2	10	-8.5
7	6	0	-8.5	**t7**	6	0	-8.5

Table 6-24 Terrace 2's ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	6	5	0	17
2	4	5	8.5	4
3	-2	5	8.5	8
4	-6	5	4.25	8.5
5	-6	5	-4.25	8.5
6	-2	5	-8.5	8
7	4	5	-8.5	4

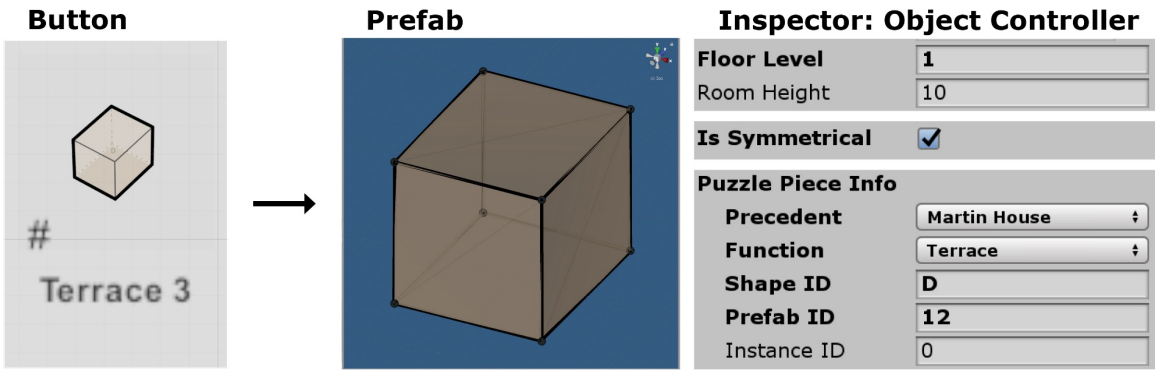


Figure 6-66 Terrace 3 Button

Source: Author

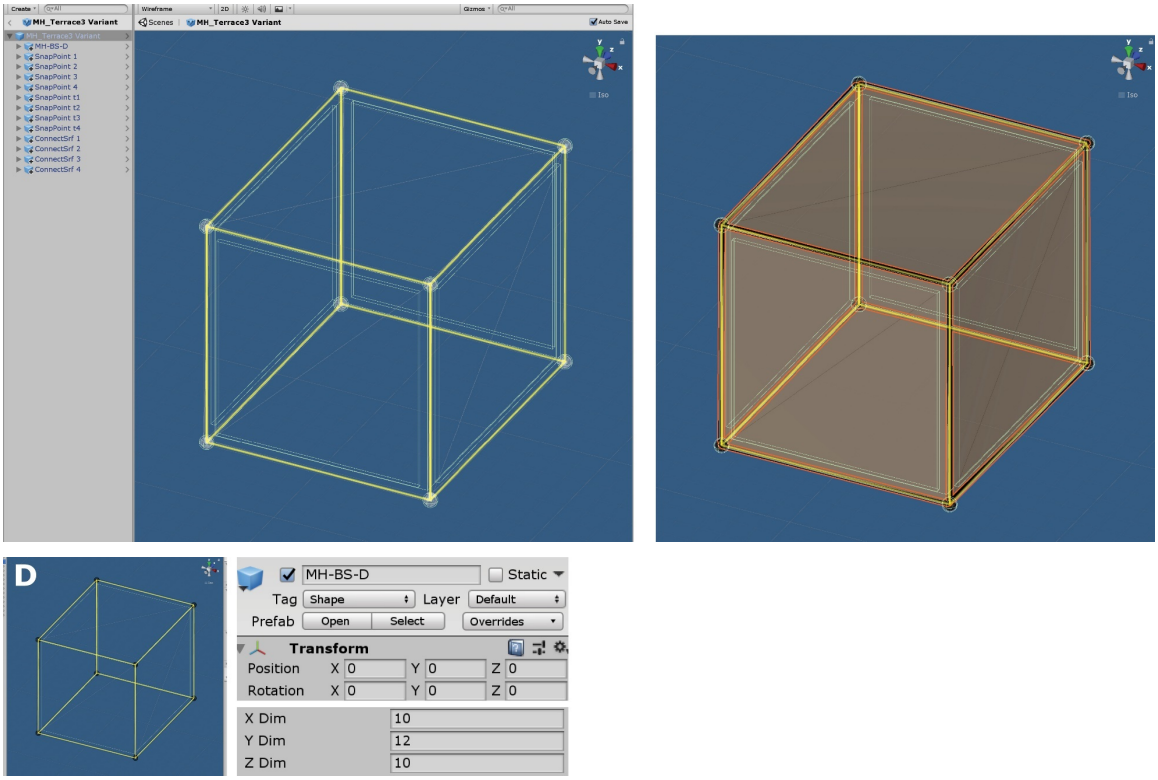


Figure 6-67 Terrace 3 Shape Composition

Source: Author

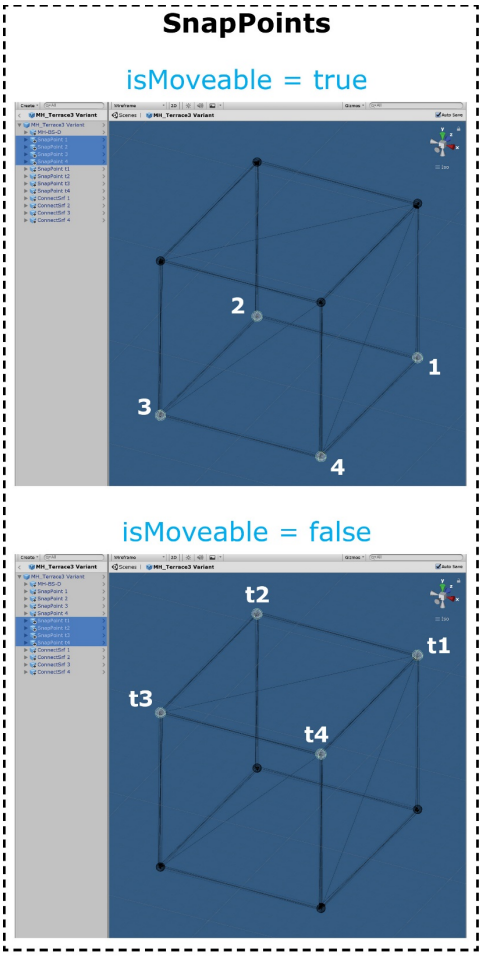


Figure 6-68 Terrace 3's SnapPoint & ConnectSrf Locations
Source: Author

Table 6-26 Terrace 3's SnapPoint Local Location Values

	Position				Position		
	x	y	z		x	y	z
1	5	0	6	**t1**	5	10	6
2	-5	0	6	**t2**	-5	10	6
3	-5	0	-6	**t3**	-5	10	-6
4	5	0	-6	**t4**	5	10	-6

Table 6-25 Terrace 3's ConnectSrf Local Location & Sizing Values

	Position			Length
	X	y	z	
1	5	5	0	12
2	0	5	6	10
3	-5	5	0	12
4	0	5	-6	10

Appendix C Martin House Scoring Criteria Lists

Table 6-27 Form Solution List Editor Inputs

Function	Shape ID	Position			Rotation			Is Sym-metrical?
(Info.Function)	*(string)*	*(Vector3)*			*(Vector3)*			*(bool)*
		x	**y**	**z**	**x**	**y**	**z**	
Fireplace	"M + M"	0f	0f	0f		0f		true
LivingRoom	"A + H"	1.5f	0f	-11.5f		0f		false
ReceptionHall	"B + B"	-15f	0f	-17.5f		0f		false
DiningRoom	"A + H"	-31.5f	0f	-11.5f		180f		false
Kitchen	"C + I + I"	-15f	0f	7f		0f		false
Bath	"K + N"	-7.5	10f	-1.25f		180f		false
Bedroom	"E + J"	-22.5f	10f	-15f		0f		false
Bedroom	"E + J"	-7.5	10f	-15f		0f		false
Bedroom	"B"	-36f	10f	-11.5f		0f		true
Bedroom	"B"	6f	10f	-11.5f		0f		true
Terrace	"B + J + J"	-15f	0f	-43.5f		0f		false
Terrace	"F + F + I"	-31f	0f	7f		0f		false
Terrace	"D"	0f	0f	-27.5f		0f		true
Vertical Circulation	"L + N"	-16.5f	0f	0.5f		0f		false
Horizontal Circulation	"G + K"	-15f	10f	-5f		0f		false

Table 6-28 Martin House Function Solutions Editor Inputs

Function	Connected Function	Total Connections
(Info.Function)	*(Info.Function)*	*int*
LivingRoom	Fireplace	1
LivingRoom	ReceptionHall	1
ReceptionHall	Terrace	2
ReceptionHall	DiningRoom	1
ReceptionHall	LivingRoom	1
ReceptionHall	VerticalCirculation	1
DiningRoom	ReceptionHall	1
DiningRoom	Kitchen	1
Bedroom	Fireplace	1
Bedroom	HorizontalCirculation	4
Bath	HorizontalCirculation	1
Kitchen	DiningRoom	1
Kitchen	Terrace	1
VerticalCirculation	ReceptionHall	1
VerticalCirculation	HorizontalCirculation	1
HorizontalCirculation	VerticalCirculation	1
HorizontalCirculation	Bath	1
HorizontalCirculation	Bedroom	4
Fireplace	LivingRoom	1
Fireplace	Bedroom	1
Terrace	ReceptionHall	2
Terrace	Kitchen	1

Bibliography

"Adding a Nested Prefab in Prefab Mode." Manual, Unity Technologies, Updated March 2018, accessed March 20, 2019, https://docs.unity3d.com/Manual/NestedPrefabs.html.

Ahmet Emre, Dincer, Çağdaş Gülen, and Tong Hakan. "A Digital Tool for Customized Mass Housing Design." Paper presented at the Fusion, Proceedings of the 32nd International Conference on Education and research in Computer Aided Architectural Design in Europe, Newcastle upon Tyne, UK, 2014 2014.

Alexander, Christopher. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

Andino, Dulce, and Chien Sheng-Fen. "Embedding Shape Grammars in a Parametric Design Software." Paper presented at the Knowledge-based Design - Proceedings of the 17th Conference of the Iberoamerican Society of Digital Graphics, Valparaiso, Chile, 2013 2013.

Andō, Tadao. "Tadao Andō : Details." In *Andō Tadao diterūshū*. Tokyo: Tokyo : A.D.A. Edita, 1991.

Buelinckx, H. "Wren's Language of City Church Designs: A Formal Generative Classification." *Environment and Planning B: Planning and Design* 20, no. 6 (1993): 645-76. https://doi.org/10.1068/b200645.

Carroll, John M. "Introduction: The Scenario Perspective on System Development." In *Scenario-Based Design : Envisioning Work and Technology in Systems Development*

New York: New York : Wiley, 1995.

Cenani, S., and G. Cagdas. "A Shape Grammar Study: Form Generation with Geometric Islamic Patterns." Paper presented at the Generative Art Conference, Milan, Italy, 2007.

Cenani, Sehnaz, and G. Cagdas. *Shape Grammar of Geometric Islamic Ornaments.* 2013.

Colakoglu, Birgul. "An Informal Shape Grammars for Interpolations of Traditional Bosnian Hayat Houses in a Contemporary Context." Paper presented at the Generative Art 2002: 5th International Generative Art Conference, Milan, Italy, 2002.

"Colliders." Manual, Unity Technologies, Updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/CollidersOverview.html.

Deterding, Sebastian, Dan Dixon, Rilla Khaled, and Lennart Nacke. "From Game Design Elements to Gamefulness: Defining "Gamification"." Paper presented at the 15th International Academic MindTrek Conference, 2011.

Duarte, José Pinto. "Towards the Mass Customization of Housing: The Grammar of Siza's Houses at Malagueira." *Environment and Planning B: Planning and Design* 32, no. 3 (2005/06/01 2005): 347-80. https://doi.org/10.1068/b31124. https://doi.org/10.1068/b31124.

Fischer, Thomas, and Christiane Herr. "Teaching Generative Design." Paper presented at the The Proceedings of the Fourth International Conference on Generative Art 2001, Milan, Italy, 2001.

Flemming, U. "The Secret of the Casa Giuliani Frigerio." *Environment and Planning B: Planning and Design* 8, no. 1 (1981): 87-96. https://doi.org/10.1068/b080087.

"Gameobjects." Manual, Unity Technologies, Updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/GameObjects.html.

Garcia, Sara, and Luis Romao. *Style and Type in a Generic Shape Grammar: The Case of Multipurpose Chairs.* 2016.

Grasl, Thomas, and Athanassios Economou. "From Shapes to Topologies and Back: An Introduction to a General Parametric Shape Grammar Interpreter." *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 32, no. 2 (2018): 208-24. https://doi.org/10.1017/S0890060417000506. https://www.cambridge.org/core/article/from-shapes-to-topologies-and-back-an-introduction-to-a-general-parametric-shape-grammar-interpreter/18193A97F837350F38ADD09B2C786763.

Guerrero, Dixie Legler. "Prairie Style : Houses and Gardens by Frank Lloyd Wright and the Prairie School." edited by Christian Korab and Frank Lloyd Wright. New York: New York : Stewart, Tabori & Chang, 1999.

Hanson, N. L. R. and Radford, and Antony D. *On Modelling the Work of the Architect Glenn Murcutt.* 1986.

Heisserman, L. "Generative Geometric Design." *Computer Graphics and Applications, IEEE* 14, no. 2 (1994): 37-45. https://doi.org/10.1109/38.267469. http://ieeexplore.ieee.org/document/267469/?reload=true.

"The Hierarchy Window." Manual, Unity Technologies, Updated March 2018, accessed July 31, 2019, https://docs.unity3d.com/Manual/Hierarchy.html.

Hillier, Bill, and Julienne Hanson. *The Social Logic of Space.* Cambridge: Cambridge : Cambridge University Press, 1984. doi:10.1017/CBO9780511597237.

Knight, T. W. "Color Grammars: Designing with Lines and Colors." *Environment and Planning B: Planning and Design* 16, no. 4 (1989): 417-49. https://doi.org/10.1068/b160417.

Knight, T. Weissman. "The Generation of Hepplewhite-Style Chair-Back Designs." *Environment and Planning B: Planning and Design* 7, no. 2 (1980): 227-38. https://doi.org/10.1068/b070227.

Knight, Terry. "Computing with Ambiguity." Article. *Environment & Planning B: Planning & Design* 30, no. 2 (2003): 165. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=9592105&site=ehost-live.

———. "Computing with Emergence." Article. *Environment & Planning B: Planning & Design* 30, no. 1 (2003): 125. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=9683517&site=ehost-live.

Knight, Terry, and George Stiny. "Making Grammars: From Computing with Shapes to Computing with Things." Article. *Design Studies* 41 (2015): 8-28. https://doi.org/10.1016/j.destud.2015.08.006. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110741765&site=ehost-live.

Koning, H., and J. Eizenberg. "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses." *Environment and Planning B: Planning and Design* 8, no. 3 (1981/09/01 1981): 295-323. https://doi.org/10.1068/b080295. http://journals.sagepub.com/doi/abs/10.1068/b080295.

Krishnamurti, R., and C. F. Earl. "Shape Recognition in Three Dimensions." *Environment and Planning B: Planning and Design* 19, no. 5 (1992): 585-603. https://doi.org/10.1068/b190585.

Krishnamurti, Ramesh, and Kui Yue. "Developing a Tractable Shape Grammar." Article. *Environment & Planning B: Planning & Design* 42, no. 6 (2015): 977-1002. https://doi.org/10.1177/0265813515610673. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110826658&site=ehost-live.

Kui, Yue. "Computation-Friendly Shape Grammars with Application to Determining the Interior Layout of Buildings from Image Data." Ph.D. Disseration, Carnegie Mellon University, 2009. http://eres.library.manoa.hawaii.edu/login?url=https://search-proquest-com.eres.library.manoa.hawaii.edu/docview/304862081?accountid=27140 (33824443).

Kui, Yue, and Ramesh Krishnamurti. "A Paradigm for Interpreting Tractable Shape Grammars." Article. *Environment & Planning B: Planning & Design* 41, no. 1 (2014): 110-37. https://doi.org/10.1068/b39107. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=94594680&site=ehost-live.

———. "Tractable Shape Grammars." Article. *Environment & Planning B: Planning & Design* 40, no. 4 (2013): 576-94. https://doi.org/10.1068/b38227. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=90116960&site=ehost-live.

Kuutti, Kari. "Work Processes: Scenarios as a Preliminary Vocabulary." In *Scenario-Based Design : Envisioning Work and Technology in Systems Development*, 25. New York: New York : Wiley, 1995.

Lee, Ji-Hyun, Michael J. Ostwald, and Ning Gu. "A Combined Plan Graph and Massing Grammar Approach to Frank Lloyd Wright's Prairie Architecture." *Nexus Network Journal* 19, no. 2 (22 February 2017 2017): 279-99.

Lee, Ji-Hyun, Hyoung-June Park, Sungwoo Lim, Sun-Joong Kim, Haelee Jung, and Mark Whiting. "A Formal Approach for the Interpretation of Cultural Content(S): Evolution of a Korean Traditional Pattern, Bosangwhamun." Paper presented at the CAAD's New Frontiers: Proceedings of the 15th International Conference on Computer-Aided Architectural Design Research in Asia, Hong Kong SAR, 2010 2010.

Lee, Ju Hyun, Michael J. Ostwald, and Ning Gu. "A Justified Plan Graph (Jpg) Grammar Approach to Identifying Spatial Design Patterns in an Architectural Style." *Environment and Planning B: Urban Analytics and City Science* 45, no. 1 (2018): 67-89. https://doi.org/10.1177/0265813516665618.

———. "A Syntactical and Grammatical Approach to Architectural Configuration, Analysis and Generation." Article. *Architectural Science Review* 58, no. 3 (2015): 189-204. https://doi.org/10.1080/00038628.2015.1015948.

http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=102747902&site=ehost-live.

"Graph Theory." Descrete Mathmatics: an Open Introduction, 2016, accessed January 21, 2019, http://discrete.openmathbooks.org/dmoi2/ch_graphtheory.html.

Linehan, Conor, George Bellord, Ben Kirman, Zachary Morford, and Bryan Roche. "Learning Curves: Analysing Pace and Challenge in Four Successful Puzzle Games." 181-90, 2014.

"Linerenderer." Manual, Unity Technologies, Updated May 31, 2017, accessed March 11, 2019, https://docs.unity3d.com/Manual/class-LineRenderer.html.

Linhares, Bruna, Helena Alarcao, Luis Carvao, Pedro Toste, and Alexandra Paio. "Using Shape Grammar to Design Ready-Made Housing for Humanized Living. Towards a Parametric-Typological Design Tool." Paper presented at the Augmented Culture: Proceedings of the 15th Iberoamerican Congress of Digital Graphics, Santa Fe, Argentina, 2011 2011.

Mitchell, William J. *The Logic of Architecture : Design, Computation, and Cognition.* Cambridge, Mass.: Cambridge, Mass. : MIT Press, 1990.

"Monobehaviour." Scripting API, Unity Technologies, Updated March, 2018, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

"Object.Instantiate." Scripting API, Unity Technologies, Updated March 13, 2019, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/Object.Instantiate.html.

Özkar, Mine. "Visual Schemas: Pragmatics of Design Learning in Foundations Studios." *Nexus Network Journal* 13, no. 1 (2011/04/01 2011): 113-30. https://doi.org/10.1007/s00004-011-0055-7. https://doi.org/10.1007/s00004-011-0055-7.

Park, Hyoung-June, and Emmanuel-George Vakaló. "A Form-Making Algorithm. Shape Grammar Reversed." Paper presented at the Conference on Computer Aided Architectural Design Futures, 2001.

Piazzalunga, U., and P. Fitzhorn. "Note on a Three-Dimensional Shape Grammar Interpreter." *Environment and Planning B: Planning and Design* 25, no. 1 (1998): 11-30. https://doi.org/10.1068/b250011.

"Prefabs." Manual, Unity Technologies, Updated July 31, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Prefabs.html.

"Quaternion." Scripting API, Unity Technologies, Updated March, 2018, accessed March 20, 2019, https://docs.unity3d.com/ScriptReference/Quaternion.html.

"Raycasts." Manual, Unity Technologies, Updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Raycasters.html.

Rogers, Ryan. "The Motivational Pull of Video Game Feedback, Rules, and Social Interaction: Another Self-Determination Theory Approach." *Computers in Human Behavior* 73 (2017): 446-50. https://doi.org/10.1016/j.chb.2017.03.048.

Sandstrom, Alice, and Hyoung-June Park. "Reflection in Action: An Educational Indie Video Game with Design Schema." CAADRIA, Wellington, New Zealand, 2019.

Sass, Lawrence. *Synthesis of Design Production with Integrated Digital Fabrication.* Vol. 16, 2007. doi:10.1016/j.autcon.2006.06.002.

———. "Wood Frame Grammar: Cad Scripting a Wood Frame House." CAAD Futures Conference, Vienna, 2005.

Sbriglio, Jacques. *Le Corbusier : The Villa Savoye.* Villa Savoye. Paris
Basel, Switzerland : Fondation Le Corbusier : Birkhäuser, 2008.

"Scenes." Manual, Unity Technologies, Updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/CreatingScenes.html.

Schön, Donald A. "The Reflective Practitioner : How Professionals Think in Action." New York: New York : Basic Books, 1983.

Shelden, Dennis R. "Digital Surface Representation and the Constructibility of Gehry's Architecture." Ph.D in Architecture: Design and Computation Thesis (Ph.D), Massachusetts Institute of Technology, 2002.

"What Game Narrative Is and What It Means in Casual Games." Medium Gaming, Updated September 13, 2018, accessed March 3, 2019, https://medium.com/@alexstargame/what-game-narrative-is-and-what-it-means-in-casual-games-67f35c191424.

Stiny, G. "Ice-Ray: A Note on the Generation of Chinese Lattice Designs." *Environment and Planning B: Planning and Design* 4, no. 1 (1977): 89-98. https://doi.org/10.1068/b040089.

———. "Introduction to Shape and Shape Grammars." *Environment and Planning B: Planning and Design* 7, no. 3 (1980): 343-51. https://doi.org/10.1068/b070343.

———. "Kindergarten Grammars: Designing with Froebel's Building Gifts." *Environment and Planning B: Planning and Design* 7, no. 4 (1980): 409-62. https://doi.org/10.1068/b070409.

———. "Weights." *Environment and Planning B: Planning and Design* 19, no. 4 (1992): 413-30. https://doi.org/10.1068/b190413.

Stiny, G., and W. J. Mitchell. "The Palladian Grammar." *Environment and Planning B: Planning and Design* 5, no. 1 (1978): 5-18. https://doi.org/10.1068/b050005.

Stiny, George. *Shape : Talking About Seeing and Doing.* Cambridge, MA: Massachusetts Institute of Technology, 2006.

"Spotlight: Frank Lloyd Wright." ArchDaily, 2018, accessed March 10, 2019, https://www.archdaily.com/513642/happy-birthday-frank-lloyd-wright.

"Tag." Manual, Unity Technologies, Updated October 12, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/Tags.html.

Tapia, M. "A Visual Implementation of a Shape Grammar System." Article. *Environment & Planning B: Planning & Design* 26, no. 1 (1999): 59. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=1546802&site=ehost-live.

"Unity Technologies Logo.Svg." 2011, accessed 11/25/2018, https://commons.wikimedia.org/wiki/File:Unity_Technologies_logo.svg.

Tepavčević, Bojan, and Vesna Stojaković. "Shape Grammar in Contemporary Architectural Theory and Design." *Facta Universitatis-series: Architecture and Civil Engineering* 10, no. 2 (2012): 169-78.

TheHappieCat. "How Game Engines Work!", 8:21, September 7 2015. Video file. https://www.youtube.com/watch?v=DKrdLKetBZE.

"Transform." Manual, Unity Technologies, Updated August 8, 2018, accessed March 11, 2019, https://docs.unity3d.com/Manual/class-Transform.html

Trescak, Tomas, Marc Esteva, and Inmaculada Rodriguez. "A Shape Grammar Interpreter for Rectilinear Forms." *Computer-Aided Design* 44, no. 7 (2012): 657-70. https://doi.org/10.1016/j.cad.2012.02.009.

Vardouli, Theodora. "Making Use: Attitudes to Human-Artifact Engagements." Article. *Design Studies* 41 (2015): 137-61. https://doi.org/10.1016/j.destud.2015.08.002. http://eres.library.manoa.hawaii.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=vth&AN=110741766&site=ehost-live.

Westwood, Dave, and Mark D. Griffiths. "The Role of Structural Characteristics in Video-Game Play Motivation: A Q-Methodology Study." *Cyberpsychology, Behavior, and Social Networking* 13, no. 5 (2010): 581-85. https://doi.org/10.1089/cyber.2009.0361.

Wright, Frank Lloyd. *Frank Lloyd Wright: The Early Work.* New York: New York, Horizon Press, 1968.